# Register Saturation in Data Dependence Graphs

Sid Touati, François Thomasset

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

# *Register Saturation in Data Dependence Graphs*

Sid-Ahmed-Ali TOUATI , François THOMASSET

**N ° 3978**

July 25, 2000

THÈME 1


ISSN 0249-6399 ISRN INRIA/RR--3978--FR+ENG

# Register Saturation in Data Dependence Graphs

Sid-Ahmed-Ali TOUATI , François THOMASSET

Thème 1 — Réseaux et systèmes
Projet A3

Rapport de recherche n ° 3978 — July 25, 2000 — 93 pages

**Abstract:** Register constraints in ILP scheduling can be taken into account during the scheduling phase of a code. The complexity of this problem is very high. In this work, we present a new approach consisting in manipulating data dependence graphs to reduce the number of "potential" values simultaneously alive without assuming any schedule. We study theoretically the exact upper-bound of the register need for all valid schedules of a code : we call this limit the **register saturation**. It is used to build a modified data dependence graph such that any schedule of this graph will verify the register constraints and avoid introducing spill code. We study the case of Direct Acyclic Graphs and then we extend it to loops intended to software pipelining schedule. Experimental study shows that many DAGs and loops do not need register constraints during scheduling.

**Key-words:** register constraints, register need, valid schedule, parallel topological sort, values simultaneously alive , set-weighted covering set, register sufficiency, register saturation

*(Résumé : tsvp)*

This work was partially supported by ESPRIT Project MHAOTEU

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
Téléphone : 01 39 63 55 11 - International : +33 1 39 63 55 11
Télécopie : (33) 01 39 63 53 30 - International : +33 1 39 63 53 30

# Saturation en Registres dans les Graphes de Dépendances de Données

**Résumé :** Le nombre limité de registres dans un processeur est une contrainte supplémentaire à l'ordonnacement de code. Ce problème devient alors très complexe. Dans ce rapport, nous présentons une nouvelle approche qui consiste à analyser les dépendances de données afin de déduire une borne maximale exacte du nombre de registres requis pour tout ordonnacement valide. Nous appelons cette limite **saturation en registres**. Si elle est inférieure au nombre de registres, alors ces contraintes ne sont pas nécessaires. Sinon, nous ajoutons des arcs de serialisation dans le graphe afin de réduire la saturation au dessous du nombre de registres. Ainsi, n'importe quelle heuristique d'ordonnacement et d'allocation aura assez de regsitres et n'introduira pas par conséquent de code de vidage. Nos expériences sur plusieurs DAGs et boucles montrent que dans la majorité des cas les contraintes de registres sont obsolètes.

**Mots-clé :** contraintes de registres, besoin en registres, ordonnancement valide, tri topologique parallèle, saturation en registres, suffisance en registres, valeurs simultanément en vie

# Contents

# List of Figures

# Chapter 1

# Introduction

In Instruction Level Parallelism (ILP) compilers, code scheduling and register allocation are two difficult tasks for code optimization. Code scheduling consists in maximizing the exploitation of the ILP offered by the precedence relation between the operations in a code. One factor that inhibits such use is the register constraints. Having a limited number of registers prohibits having an unbounded number of values simultaneously alive [1]. Two major negative register constraints are:

1. serialization of independent operations, because there are not enough registers that could keep all the intermediate results ;
2. introducing spill code, that is we use the memory as an intermediate storage instead of registers.

Register allocation consists in associating a physical register to a value produced within the code. If this task is carried out before the scheduling, it introduces new false dependencies (anti and output dependencies) between the operations and limiting the amount of ILP. If carried out after, spill code might be introduced, since the scheduler tries to exploit at best the ILP, producing then more values simultaneously alive than the amount of physical registers. Spill code dramatically decreases the performance because of memory hierarchy access latencies.

A best approach is combining code scheduling under resource constraints. The relation between the two phases are studied in a lost of work [Bra94, GH88, BEH91, Pin93]. Their purpose is to try constructing a schedule with a limited amount of values simultaneously alive in order to avoid spill code when allocating registers. The main problem with such an approach is its high complexity. Only code scheduling problem under resource constraints is NP-complete. Adding register constraints to this problem increases the complexity.

In this report, we present our contribution in the field of avoiding and reducing spill code that could be generated by any schedule of a Direct Acyclic Graph (DAG). Our approach is to modify and extend the original graph in such a way that all valid schedules of the new graph verify the register constraints and avoid spilling, see figure 1. We proceed by studying the worst case in register need. We call this notion the **register saturation** of a code which is the maximal number of registers (values simultaneously alive ) that could be used by all schedules. If we succeed in limiting the register saturation, then any schedule could not exceed the register requirement. Reducing the register saturation involves introducing some new serialization arcs

[1] intermediate results of operations used to pursue the computing

in the graph restricting the instruction level parallelism to avoid spill code. We give the case when the spill code could not be avoided. However, since our approach reduces the register saturation even if it exceeds the amount of registers, any schedule would have a limited amount of excessive values simultaneously alive . Consequently, the amount of spill code is limited.

There are two major reasons for dissociating register constraints before code scheduling. First, if the register saturation does not exceed the amount of registers, we can ignore register constraints for code scheduling and we then reduce the complexity. Second, register constraints models in modern processors are similar because there is few possible configurations : the amount of registers can be 32, 64 or 128, and the register types are classified as integer or float. However, resource constraints are less comparable : each processor can have its own properties. This obliges us to redo code scheduling for each target processor. With our approach, we can build a DAG that satisfies register constraints for a whole family of processors. For instance, if we construct a DAG with a register saturation of 32, we can schedule it in any modern target processors that have not less than 32 registers.

This document contains the following sections. Chapter 2 recalls some basic theoretical notions and notations in graphs. Chapter 3 presents our work on the formulation and the resolution of the register saturation problem in the case of an acyclic graph (DAG) : the first section gives a formulation of all valid schedules for a DAG. Then we present how to deduce the family of DAGs that saturate the register pressure. Afterwards, we give our solution to reduce the register saturation. Chapter 4 extends our work to the case of a loop with possible cyclic graphs. We have developed our approach and we give our experimental results in chapter 5. Chapter 6 presents some related work in this field. We conclude by our remarks and perspectives in chapter 7.

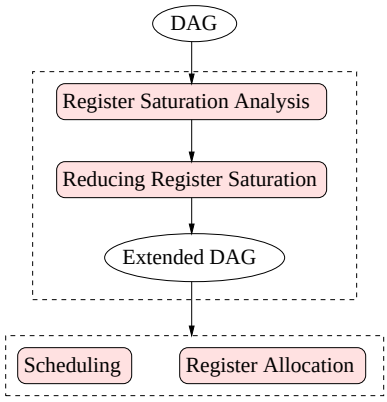



Figure 1.1: Register saturation analysis steps

# Chapter 2

# Theoretical Background and Notations

This chapter only recall some notations and definitions that are used in this report. To have an complete overview of the theory, the reader should refer to standard books.

## 2.1 Graphs and Hypergraphs

An oriented **graph** $G = (V, E)$ is a couple of a set $V$ and a binary relation $E \subseteq V^2$. Let $u \in V$, then we define the following notations :

- $u \in V$ is called an **node** ;
- $e = (u, v) \in E$ is called an **arc** ;
- $\forall e = (u, v) \in E$, $u$ (resp. $v$) is called the **source** (resp. the **target**) of the arc $e$. Both $u$ and $v$ are called the **endpoints** of $e$ ;
- $\Gamma^+_G(u) = \{v \in V / (u, v) \in E\}$ the set of the $u$'s successors ;
- $\Gamma^-_G(u) = \{v \in V / (v, u) \in E\}$ the set of the $u$'s predecessors ;
- $d^+_G(u) = |\Gamma^+(u)|$ the **outdegree** of $u$ ;
- $d^+_G(u) = |\Gamma^-(u)|$ the **indegree** of $u$ ;
- if $d^-_G(u) = 0$ then $u$ is called a source of $G$ ;
- if $d^+_G(u) = 0$ then $u$ is called a **sink** of $G$ ;
- $Source(G) = \{u \in V / d^-(u) = 0\}$ ;
- $Sink(G) = \{u \in V / d^+(u) = 0\}$ ;
- $\forall e = (u, v) \in E : source(e) = u$ and $target(e) = v$ ;
- a **path** in $G$ is a k-tuple $p = (e_1, \ldots, e_k) \in E^k$ such that $\forall i = 1, ...k \ : \ target(e_i) = source(e_{i+1})$ ;
- a **circuit** in $G$ is a path $p = (e_1, \ldots, e_k) \in E^k$ such that $\forall i = 1, ...k \ : \ target(e_i) = source(e_{i+1}) \wedge target(e_k) = source(e_1)$.

- two nodes $u, v$ are **adjacent** *iff* there is an arc connecting them :

$$\exists e \in E \qquad \{u, v\} = enpoints(e)$$

- two arcs $e, e'$ are **adjacent** *iff* there is a shared node between them ;

$$\exists u \in V \qquad endpoints(e) \cap endpoints(e) = \{u\}$$

A $G = (V, E)$ is a **complete graph** *iff* $(u, v) \notin E \Longrightarrow (v, u) \in E$.
A **subgraph** $G_{V'}$ induced by $V' \subseteq V$ is the graph that contains all nodes of $V'$ and the arcs that have their endpoints in $V'$. We note also $G_{V-V''} = G - V''$ for $V'' \subseteq V$.
A **partial graph** $G'$ of $G = (V, E)$ generated by a subset $E' \subseteq E$ is the graph that contains all the nodes of $G$ but only the arcs contained in $E'$. That is we eliminate all the arcs in $E - E'$. We write $G' = G/_{E'}$ .

For the need of this report, we introduce the concept of **extended graph**. An extended graph is only the dual definition of a partial graph. An extended graph $G'$ of $G = (V, E)$ generated by a subset $E' \subseteq V^2$ is the graph that contains all the nodes of $G$ and the arcs in $E$ extended by the arcs contained in $E'$. That is we only add all the arcs in $E'$. We write $G' = G\backslash^{E'}$.

The **transitive closure** of $G$, noted $G_c$ , is an extended graph $G\backslash^{E_c}$ such that :

$$E_c = \{(u_1, u_2)/(u_1, u_2) \in V^2 \wedge \exists \text{ a path } p = (u_1, \ldots, u_2)\}$$

That is we only add all transitive arcs.

The **transitive reduction** of $G$, noted $G_r$ , is a partial graph $G/_{E_r}$ such that :

$$E_r = \{e = (u_1, u_2) \in E/\forall \text{ path } p = (u_1, \ldots, u_2) \Longrightarrow p = (u1, u2)\}$$

That is we only remove transitive arcs.

### 2.1.1 Some Notions for Directed Acyclic Graphs

Let $G = (V, E)$ be a DAG. A topological sort (called also **linear extension** ) of $G = (V, E)$ is a permutation $(u_1, u_2, \ldots, u_n)$ of $V$'s elements such that

$$(u_i, u_j) \in E \Longrightarrow i < j$$

We use the concatenation symbol "$\cdot$" to write a linear extension as $u_1 \cdot u_2 \cdots u_n$. Let $\mathcal{L}(G)$ be the set of all linear extensions of $G$. We can write

$$\mathcal{L}(G) = \bigcup_{u \in Source(G)} \Big\{ u \cdot \sigma / \sigma \in \mathcal{L}\big(G - \{u\}\big) \Big\}$$

The transitive closure of a DAG defines the notion of parallel and comparable :

- $\forall u, v \in V : u \sim v \Longleftrightarrow (u, v) \in E_c \vee (v, u) \in E_c$. $u$ and $v$ are said **comparable** ;
- $\forall u, v \in V : u||v \Longleftrightarrow \neg(u \sim v)$. $u$ and $v$ are said **parallel** ;

- $\forall u, v \in V : u < v \Longleftrightarrow (u, v) \in E_c$ that is $<$ defines an order between the nodes.
- $\forall u, v \in V : u \leq v \Longleftrightarrow u = v \vee u < v$

We define also the notions of **descendants** and **ascendants** of a node $v \in V$ :

- $\uparrow v = \{u \in V / u \leq v\}$ the set of $v$'s ascendants including $v$ ;
- $\downarrow v = \{u \in V / v \leq u\}$ the set of $v$'s descendants including $v$.

We define the notion of chain and antichain in an acyclic graph :

- A subset $C \subseteq V$ in $G = (V, E)$ is a **chain** *iff* : $\forall u, v \in C : u \sim v$
- A chain $MC$ is said **maximal** *iff* $\forall C$ a chain $: |C| \leq |MC|$
- A subset $A \subseteq V$ in $G = (V, E)$ is an **antichain** *iff* : $\forall u, v \in A : u \parallel v$
- An antichain $MA$ is said **maximal** *iff* $\forall A$ an antichain $: |A| \leq |AM|$

### 2.1.2 Hypergraphs

An **hypergraph** $H = (S, \mathcal{E})$ is a couple of two sets : $S = \{s_1, s_2, ...., s_n\}$ and a family $\mathcal{E} = \{E_1, E_2, ..., E_m\}$ of subsets from $S$ such that :

$$\forall j = 1, m : E_j \neq \phi$$

and

$$\bigcup_{j=1,m} E_j = S \text{ (covering constraint)}$$

The elements $s_1, s_2, ...., s_n$ are the nodes of the hypergraph, and the subsets $E_1, E_2, ..., E_m$ are called the **edges** . Graphically, an hypergraph $H = (S, \mathcal{E})$ is represented by joining the nodes such that $\forall s_i \in E_j$ :

- if $|E_j| = 1$ then put a loop joined on the node ;
- if $|E_j| = 2$ then we join the two nodes by a line (like in an indirected graph) ;
- if $|E_j| > 2$ then we surround all nodes by a joined line.

## 2.2 Interval Orders

In this report, we use the 13-values interval algebra studies in [GS92]. Let $I_1 = [a_1, b_1] \in \mathbb{N}$ and $I_2 = [a_2, b_2] \in \mathbb{N}$ be two intervals. Then :

1. $I_1 \prec I_2 \Longleftrightarrow b_1 < a_2$. We says that $I_1$ is before $I_2$ ;
2. $I_1 \, f \, I_2 \Longleftrightarrow b_1 = b_2$. We says that $I_1$ finishes $I_2$.

# Chapter 3

# Case of DAGs

In this chapter, we study the notion of *register saturation* in a DAG, which is the maximum number of registers that can use a given acyclic graph to complete the computation. The notion of register in our report is an alive value. The number of registers consumed is then the number of values simultaneously alive . To avoid ambiguity between values simultaneously alive and processor registers, we refer to the latter as physical registers in the entire report.

Section 3.1 presents our formal definition of a valid schedule. We extend the notion of linear extension of a graph to formalize horizontal schedules (Very Large Instruction Word). Section 3.2 studies the problem of finding an extended graph of an original one such that the register consumption of any schedule for this graph is maximal. Section 3.3 studies the problem of finding an extended graph of the original one such that the register consumption of any schedule for this DAG is limited by a constant.

## 3.1 Valid Schedules

We begin by introducing the notion of parallel topological sort of a DAG that models horizontal schedules, such that parallel nodes are written in the same slot. The standard topological sort is referred to as *strict* topological sort, because we can only put one node per slot. In a parallel topological sort, we can put more than one operation per slot.

**Definition 3.1 (Parallel Topological Sort of a DAG)** *A parallel topological sort of an acyclic graph $G = (V, E)$, is a family $\mathcal{T}$ of ordered non empty subsets of $V$ $(V_0, V_1, \ldots, V_m)$ with $0 \leq m < |V|$ such that:*

- *each node belongs to one and only one subset*

$$\forall u \in V, \exists! V_i \in \mathcal{T} : u \in V_i$$

*We note $line_{\mathcal{T}}(u) = i$ and $slot_{\mathcal{T}}(u) = V_i$*

- *precedence constraints*

$$\forall e = (u, v) \in E \qquad line_{\mathcal{T}}(u) < line_{\mathcal{T}}(v)$$

We use the concatenation symbol "." to note a parallel topological sort as $V_0 \cdot V_1 \cdots V_m$.

Let $\mathcal{LL}$ be the set of all parallel topological sorts of $G = (V, E)$ . $\mathcal{LL}$ is computed by examining all combinations of parallel nodes. Formally, we state :

$$\mathcal{LL}(G) = \bigcup_{\substack{p=1,|Source(G)| \\ S \in C^p(Source(G))}} \{S \cdot \sigma / \sigma \in \mathcal{LL}(G - S)\}$$

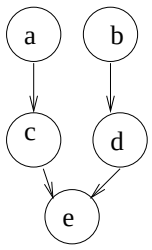
such that $C^p(V)$ defined for a set $V$ is all subsets from $V$ obtained by all combinations of $p$ elements from $V$. That is $C^p(V) = \{S \subseteq V / |S| = p\}$ and

$$|C^p(V)| = C^p_{|V|} = \frac{|V|!}{p! \times (|V| - p)!}$$

Figure 3.1 gives some examples of parallel topological sorts.

**Proposition 3.1** $\forall G = (V, E)$ *a DAG* : $G' = G\backslash^{E'}$ *is an extended DAG of* $G \Longrightarrow \mathcal{LL}(G') \subseteq \mathcal{LL}(G)$

**Proposition 3.2** $\forall G = (V, E)$ : *G is a DAG* $\Longrightarrow \mathcal{LL}(G) = \mathcal{LL}(G_c) = \mathcal{LL}(G_r)$




(a) a DAG

| | | |
|---|---|---|
| a | b | a |
| b | d | c |
| c | a | b |
| d | c | d |
| e | e | e |

(b) strict topological sorts

| | | |
|---|---|---|
| {a, b} | a | a |
| {c, d} | b | c |
| e | {c,d} | b |
| | e | d |
| | | e |

(c) parallel topological sorts

Figure 3.1: Examples of strict an parallel topological sorts

**Definition 3.2 (Parallel schedules of a DAG)** *A valid schedule* $\sigma$ *of a DAG* $G = (V, E)$ *associated to a parallel topological sort* $\mathcal{T} \in \mathcal{LL}(G)$ *is a function such that :*

$$\begin{array}{rccl} \sigma : & V & \rightarrow & N \\ & v & \mapsto & line_{\mathcal{T}}(v) \end{array}$$

We note $\Sigma(G)$ the set of all valid schedules. As $0 \leq m < |V|$, we can state that

$$\forall \sigma \in \Sigma(G), \forall v \in V : 0 \leq \sigma(v) < |V|$$

Appendix A gives an algorithm for computing $\Sigma(G)$

### 3.1.1 Restricted Schedules

In this section, we reduce the definition of parallel topological sorts by introducing a upper limit on the number of parallel nodes contained in one slot. This concept models the maximum exploitable ILP by a processor.

**Definition 3.3 (Restricted parallel topological sort of a DAG)** *A restricted parallel topological sort of an acyclic graph $G = (V, E)$ with a non negative integer $r$, is a parallel topological sort noted $\overline{\mathcal{T}}^r = V_0 \cdot V_1 \cdots V_m$ with $0 \leq m < |V|$ such that: $\forall V_i \in \overline{\mathcal{T}}^r : |V_i| \leq r$*

We note $\overline{\mathcal{LL}}^r(G) \subseteq \mathcal{LL}(G)$ the set of all restricted parallel topological sorts of $G$ with the non negative integer $r$. $\overline{\mathcal{LL}}^r(G)$ is computed by examining all combinations of $r$ parallel nodes. Formally, we write :

$$\overline{\mathcal{LL}}^r(G) = \bigcup_{\substack{p=1,min(r,|Source(G)|) \\ S \in C^p(Source(G))}} \{S \cdot \sigma / \sigma \in \overline{\mathcal{LL}}^r(G - S)\}$$

**Definition 3.4 (Restricted valid schedule of a DAG)** *A valid schedule $\bar{\sigma}^r$ of a DAG $G = (V, E)$ is restricted by a non negative integer $0 < r \leq |V|$ iff :*

$$\exists \overline{\mathcal{T}}^r \in \overline{\mathcal{LL}}^r(G), \forall v \in V : \bar{\sigma}^r(v) = line_{\overline{\mathcal{T}}^r}(v)$$

We note $\overline{\Sigma}^r(G) \subseteq \Sigma(G)$ the set of all restricted valid schedules of $G$ with the non negative integer $r$.

**Remark** The strict topological sorts are restricted topological sorts with limit 1 :

$$\mathcal{L}(G) = \overline{\mathcal{LL}}^1(G)$$

### 3.1.2 Schedules with NUAL Semantics

Until now, $\Sigma(G)$ the set of all valid schedules defined for a DAG $G = (V, E)$ models all schedules with the UAL [1] semantics [SRM94]. In the case of UAL semantic, precedence constraints are sufficient to build valid schedules. Building a valid schedule in this semantic is to schedule one operation **before** its descendants. Operation latencies are not important for the validity. But with NUAL[2] semantic, like in VLIW processors, $\Sigma(G)$ does not model valid schedules since it does not guarantee the operation latencies between nodes. Scheduling one operation before its descendants is not sufficient for building a valid schedule : we must guarantee a minimum of time steps which corresponds to the latency of the operation.

For this purpose, we introduce in $G$ the function $\delta$ that gives the **latency** of each arc. We define $G = (V, E, \delta)$ such that $\forall e \in E : \delta(e)$ is the latency of $e$. A valid schedule $\sigma$ of $G$ must satisfy

$$\forall e = (u, v) \in E : \sigma(v) - \sigma(u) \geq \delta(e)$$

To model this sort of semantics, we should redefine the parallel topological sort (see figure 3.2).

[1] Unit Assumed Latency : the processor has to guarantee the data dependence if an operation is scheduled before another. For the scheduler, all operations have a unit latency.

[2] Non Unit Assumed Latency : the scheduler has to guarantee the data dependence by inserting *nops* between operations if the latency is more than 1 cycle.

**Definition 3.5 (NUAL Parallel Topological Sort of a DAG)** *A NUAL parallel topological sort of an acyclic graph $G = (V, E)$ , is a family $\mathcal{T}$ of ordered subsets of $V$ $(V_0, V_1, \ldots, V_m)$ such that:*

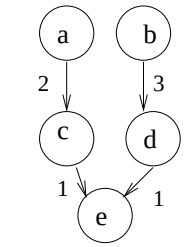
- *each node belongs to one and only one subset*

$$\forall u \in V, \exists! V_i \in \mathcal{T} : u \in V_i$$

*We note $line_{\mathcal{T}}(u) = i$ and $slot_{\mathcal{T}}(u) = V_i$*

- *each precedence constraint must be guaranteed with its minimal latency*

$$\forall (u, v) \in E : line_{\mathcal{T}}(u) + \delta(e) \leq line_{\mathcal{T}}(v)$$




(a) a NUAL DAG

| | | |
|---|---|---|
| {a, b} | {a} | {a} |
| {} | {b} | {} |
| {} | {c} | {b, c} |
| {c, d} | {} | {} |
| {e} | {d} | {} |
| | {e} | {d} |
| | | {e} |

(b) NUAL parallel topological sorts

Figure 3.2: NUAL parallel topological sorts

Let be $\mathcal{LL}_{NUAL}$ the set of all NUAL parallel topological sorts. To compute this set, we define a graph transformation $\varphi$ that introduces new virtual nodes (*nops*) and arcs in $G$ in order to guarantee the minimum arc latency. Let $D_{UAL}$ be the set of all DAGs defined with UAL semantic (precedence constraints are sufficient to build a valid schedule), and $D_{NUAL}$ the set of all DAGs in the NUAL semantic (a valid schedule should guarantee the $\delta$ latencies).

$$D_{UAL} = \{G = (V, E) \ / G \text{ is a DAG}\}$$

$$D_{NUAL} = \{G = (V, E, \delta) / G \text{ is a DAG}\}$$

Now we give the definition of our graph transformation.

**Definition 3.6 (NUAL Transformation)** *A NUAL transformation $\varphi$ defined for any DAG $G \in N_{UAL}$ is a function such that*

$$\begin{array}{rlll} \varphi : & D_{NUAL} & \rightarrow D_{UAL} \\ & G = (V, E, \delta) & \mapsto \varphi(G) = (V', E') \end{array}$$

*with*

$$\begin{array}{rlll} V' = & V & \cup \{nop_{u,i} / u \in V \wedge 0 \leq i \leq \max_{e=(u,v)\in E} \delta(e) - 2\} \\ E' = & E & \cup \{(nop_{u,i}, nop_{u,i+1}) / u \in V \wedge 0 \leq i \leq \max_{e=(u,v)\in E} \delta(e) - 2\} \\ & & \cup \{(u, nop_{u,0}), \forall u \in V\} \\ & & \cup \{(nop_{u,\delta(e)-2}, v) / e = (u, v) \in E\} \\ & & - \{e = (u, v) \in E / \delta(e) > 1\} \end{array}$$

$V'$ is extended by a chain of intermediate nops for any node $u$ in order to introduce the minimum latency $\delta(e)$. If the minimum latency is equal to 1, then no *nop* are introduced. In $E'$, we connect each node $u$ with its first intermediate nop, and we connect an intermediate *nop* to the $u$'s successor depending on the latency of arc $e$. Figure 3.3 gives an example of a NUAL transformation.
Now, the set of all valid schedules of a DAG in NUAL semantics is:

$$\forall G \in D_{NUAL} : \Sigma(G) = \Sigma\big(\varphi(G)\big)$$

**Remark** theoretically, there is an **infinite number of schedules** for a given DAG $G \in D_{NUAL}$. That is because we can insert an infinite number of *nops* to guarantee a data dependence between two operations. By using the NUAL transformation, we restrict the infinite set of valid schedules to the finite set $\Sigma\big(\varphi(G)\big)$.

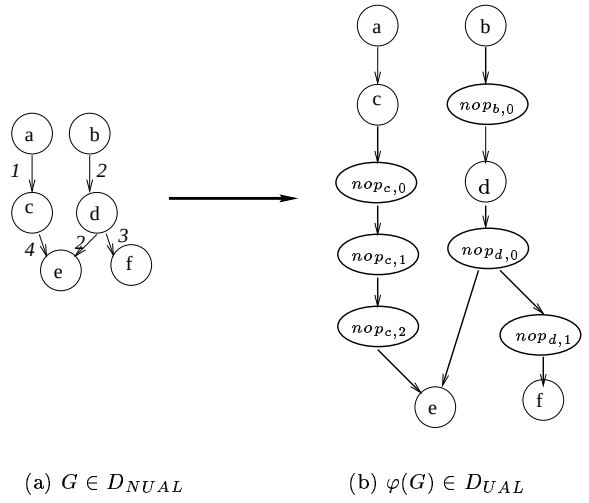
## 3.2 Register Saturation Problem

In this section, we study how to find a family of schedules maximizing the register utilization for a given DAG. We will see that this problem is reduced to answering the question *which operation kills which value ?* In this report, the notion of register is a **value alive**. To avoid ambiguity, we refer to a processor's register as a *physical* register.

### 3.2.1 DAG Model

A DAG $G = (V, E, \delta, \delta_w, \delta_r)$ in our study represents data dependence constrains within operations, such that:

- $V$ is the operations set;
- $E = \{(u, v)/\ u, v \in V\}$ are data dependence constraints;
- $\forall e \in E$, $\delta(e)$ is a positive integer representing the source operation latency.




(a) $G \in D_{NUAL}$ (b) $\varphi(G) \in D_{UAL}$

Figure 3.3: NUAL transformation

Since writing and reading into and from registers could be delayed from the beginning of the operation schedule time (NUAL semantics, like VLIW and EPIC), we define the two delay functions $\delta_r$ and $\delta_w$:

$$\begin{array}{ll} \delta_r : & V \rightarrow \mathbb{N} \\ & u \mapsto \delta_r(u) \text{ cycle number when the read occurs after } \sigma(u) \\ \delta_w : & V_R \rightarrow \mathbb{N} \\ & u \mapsto \delta_w(u)/\delta_r(u) \leq \delta_w(u) \text{ cycle number when the write occurs after } \sigma(u) \end{array}$$

Then each operation $u$ read from registers at instant $\sigma(u) + \delta_r(u)$, and each value $v$ is written at instant $\sigma(v) + \delta_w(v)$. Each operation $u \in V$ has a strictly positive latency $lat(u)$, so in the initial DAG we have $\forall e = (u, v) \in E \ \delta(e) = lat(u)$. We suppose also that the read occurs at the beginning of the operation latency and the write at the end of the latency: $\delta_r(u) \leq \delta_w(u) < lat(u)$.

When studying register need in a DAG representing a data dependence graph (DDG) and some other precedence relations, we should make a difference between the nodes, depending if they write in a physical register or not. Depending on which register type we are focusing on (int, real), we set $V = V_R \cup V_S$ for any $G = (V, E, \delta, \delta_w, \delta_r)$ such that:

- $V_R$ are the subset of operations that write into one physical register of the considered type[3]. We call them **value nodes**[4];
- $V_S = V - V_R$ the remainder of operations (like *store, nop* ... ).

Depending on which register type we are focusing on (int, real), we also make a difference between arcs, depending if they refer to true dependence through a physical register of the considered type, or simply to a serialization (true dependence through memory or through other register types). We define $E = E_R \cup E_S$ such that:

- $E_R$ is the subset of arcs that are true dependences through a physical register of the considered type. We call them **flow** arcs;
- $E_S = E - E_R$ the remainder of arcs (like true dependence through memory or through another register type). We call them **serial** arcs.

It is clear that $(u, v) \in E_R \Longrightarrow u \in V_R$. To simplify writing our formulas, we assume that the DAG has one source and one sink that we call $\top$ and $\bot$. If not, we introduce the two virtual nodes $(\top, \bot)$ representing nops. We add an arc from $\top$ to each source with a null latency, and an arc from each sink to $\bot$ with the latency of the sink operation.

Then, a valid schedule $\sigma$ of $G$ is a function that gives an integer execution time for each operation:

$$\sigma \text{ is valid} \Longleftrightarrow \forall e = (u, v) \in E \quad \sigma(v) - \sigma(u) \geq \delta(e)$$

and we note $\Sigma(G)$ the set of all valid schedules for $G$. And we note $\overline{\sigma} = \sigma(\bot)$ the last execution step.

[3] we assume a RISC operation model, i.e. a value operation does not write into more than one register

[4] we also assume that there is one possible definition per value

**Remark** if $G' = G\backslash^{E'}$ is an extended DAG of $G$, then

$$\Sigma(G') \subseteq \Sigma(G)$$

A value produced by a node $u$ is alive between the execution of $u$ and its last consumption. We first define the consumer set of a value as the set of all operations that read this value. The values that are not read in $G$ are those that are still alive when exiting the computation and must be kept in registers. We model these special values by considering that the bottom node $\perp$ consumes these leaf values.

**Definition 3.7 (Consumer Set)** *Given DAG $G = (V, E, \delta, \delta_w, \delta_r)$, the consumer set $Cons(u)$ of a value $u \in V_R$ is defined by :*

$$Cons(u) = \begin{cases} \{v \in \Gamma^+_G(u)/(u,v) \in E_R\} & \text{if } \exists (u,v) \in E_R \\ \perp & \text{otherwise} \end{cases}$$

Some operations in the consumer set could be killing operations (the last scheduled operations within the consumer set). To know which operation could *potentially* kill which value, we eliminate consumers that can never kill this value : they are the consumers that have precedence constraints with other consumers. We give the following definition.

**Definition 3.8 (Potential Killing Operations)** *Given a DAG $G = (V, E, \delta, \delta_w, \delta_r)$, potential killing operations of a value $u \in V_R$ is the subset $pkill(u) \subseteq Cons(u)$ such that :*

$$pkill(u) = \left\{ v \in Cons(u) / \ \downarrow v \cap Cons(u) = \{v\} \right\}$$

**Property :** All potential killing operations of a value $u$ are parallel, i.e. :

$$\forall v, v' \in pkill(u) \qquad v || v'$$

To model the potential killing relation among nodes, we use :

**Definition 3.9 (Potential Killing DAG)** *Given a DAG $G = (V, E, \delta, \delta_w, \delta_r)$, the potential killing DAG of $G$, noted $PK(G) = (V, E_{PK})$ is the partial graph $G/_{E_{PK}}$ such that :*

$$\forall u, v \in V : (u,v) \in E_{PK} \Longleftrightarrow u \in V_R \wedge v \in pkill(u)$$

Constructing $PK(G)$ can be done with polynomial complexity which is the complexity of constructing the transitive closure $G_c$ of $G$. See appendix B for more details.

**Example 3.2.1** *The DAG presented in figure 3.4.a is an example of a potential killing DAG. Bold circles are value nodes, and bold arcs are flow arcs. Part (b) models the pkill relations between value nodes and other nodes. In this example, we have for instance Cons(a)={c,d}, pkill(a)={d}, Cons(e)={g,h,i}, pkill(e)={h,i}.*

Given a schedule, the killing date is the last time when a value is consumed.

**Definition 3.10 (Killing Date of a Value)** *Given DAG $G = (V, E, \delta, \delta_w, \delta_r)$ and a schedule $\sigma \in \Sigma(G)$, we associate to each value $u \in V_R$ a killing date, noted $kill_\sigma(u)$ and defined by*

$$\forall u \in V_R \quad kill_\sigma(u) = \begin{cases} \max_{v \in pkill(u)} \sigma(v) + \delta_r(v) & \text{if } pkill(u) \neq \phi \\ \overline{\sigma} & \text{otherwise} \end{cases}$$

*We call $]\sigma(u) + \delta_w(u), \sigma(v) + \delta_r(v)]$ the* **lifetime interval** *of $u$, noted $L^\sigma_u$*

We call a **killing operation** each operation $v \in pkill(u)$ scheduled at time $kill_\sigma(u)$.

**Proposition 3.3** *Given a DAG* $G = (V, E, \delta, \delta_w, \delta_r)$, *then* $\forall u \in V_R$

$$\forall \sigma \in \Sigma(G) \quad \exists v \in pkill(u) : \sigma(v) + \delta_r(v) = kill_\sigma(u) \tag{3.1}$$

$$\forall v \in pkill(u) \quad \exists \sigma \in \Sigma(G) : \quad kill_\sigma(u) = \sigma(v) + \delta_r(v) \tag{3.2}$$

**Proof :**

Proving (3.1) is deduced directly from *pkill* definition. Since

$$v \in pkill(u) \Longrightarrow \nexists v' \in Cons(u) \quad v < v'$$

then the killing date of $u$ must be the schedule of some operation in $pkill(u)$. Let us prove that

$$\forall u \in V_R \; \nexists v' \in Cons(u) - pkill(u) \; \exists \sigma \in \Sigma(G) \quad kill_\sigma(u) = \sigma(v') + \delta_r(v')$$

Suppose the contrary true.

$$\exists v' \in Cons(u) - pkill(u) \Longrightarrow \exists v \in pkill(u)/v' < v$$

let $lp(v', v)$ be the longest path from $v'$ to $v$.

$$\text{since } lp(v', v) \geq lat(v') > \delta_r(v') \Longrightarrow \sigma(v) - \sigma(v') > \delta_r(v')$$

Since $\delta_r(v) \geq 0$:

$$\sigma(v) + \delta_r(v) - \sigma(v') > \delta_r(v') \Longrightarrow \sigma(v) + \delta_r(v) > \sigma(v') + \delta_r(v')$$

Then

$$kill_\sigma(u) \geq \sigma(v) + \delta_r(v) > \sigma(v') + \delta_r(v')$$

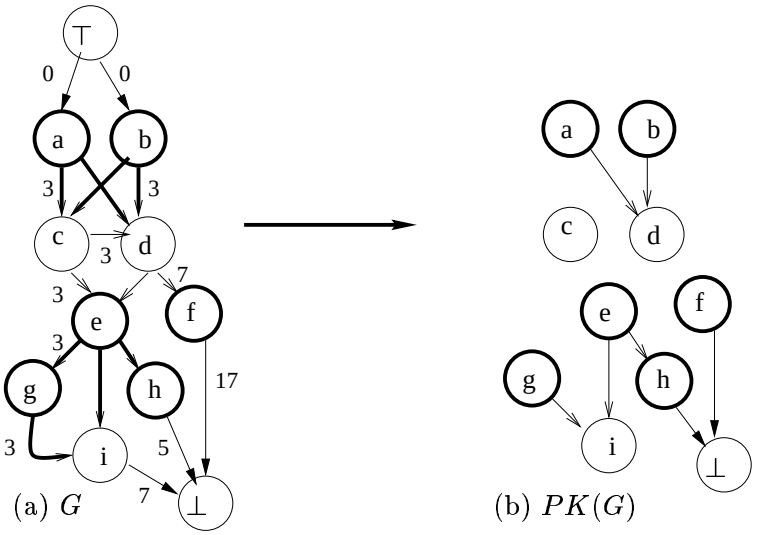
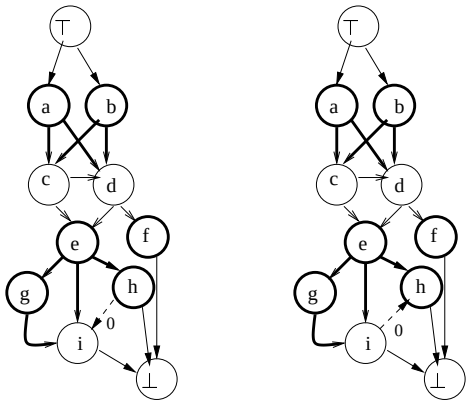



Figure 3.4: Potential killing DAG

For proving (3.2), for each $v \in pkill(u)$, we create an extended DAG $G^u_v = G\backslash^{E'}$ to enforce $v$ to be the last read of the value $u$. $\forall v' \in pkill(u) - \{v\}$, we add a serial arc $e$ from $v'$ to $v$ with latency $\delta(e) = \delta_r(v') - \delta_r(v)$ Then, any schedule $\sigma \in \Sigma(G^u_v)$ ensures $\sigma(v) + \delta_r(v) \geq \sigma(v') + \delta_r(v')$ which means $kill_\sigma(u) = \sigma(v)$. Let's prove that $G^u_v$ is still a DAG. Suppose the contrary is true, i.e. $\exists u \in V_R$, $\exists v \in pkill(u)$ such that $G^u_v$ is cyclic. Let $\mathcal{C} = (v, \cdots, v', v)$ be this cycle where the introduced arc is $(v', v)$. We know that all the potential killing operations $pkill(u)$ of a value $u$ are parallel in $G$. But before introducing this arc, a path $p = (v, \cdots, v')$ means that $v < v'$ in $G$ which is a contradiction.

Figure 3.5 shows the two extended DAGs associated to $e$. The original DAG is presented in Fig. 3.4. Here, we assume that all read delay are null. $e$ has two potential killing operations $\{h, i\}$, so we have two extended DAG : $G^e_i$ ensures that $i$ kills $e$, and $G^e_h$ that ensures that $h$ kills $e$.




(a) $G^e_i$ (b) $G^e_h$

Figure 3.5: Proposition: each potential killing operation can kill the value

⌟

The register need of a schedule is the maximum number of simultaneously alive values. It defines the amount of physical registers required to avoid spilling.

**Definition 3.11 (Register Need of a Schedule)** *Given a DAG $G = (V, E, \delta, \delta_w, \delta_r)$, the register need $RN_\sigma(G)$ of $G$ according to a schedule $\sigma \in \Sigma(G)$ is defined by*

$$RN_\sigma(G) = \max_{0 \leq i \leq \overline{\sigma}} |vsa_\sigma(i)|$$

*such that*

$$vsa_\sigma(i) = \{u \in V_R / i \in L^\sigma_u\} \text{ values simultaneously alive at instant } i$$

The register saturation is the maximal register need that can be obtained for a given DAG.

**Definition 3.12 (Register Saturation of a DAG)** *Given DAG $G = (V, E, \delta, \delta_w, \delta_r)$, the register saturation $RS(G)$ of $G$ is defined by*

$$RS(G) = \max_{\sigma \in \Sigma(G)} RN_\sigma(G)$$

**Definition 3.13 (Saturation Schedule)** *Given DAG $G = (V, E, \delta, \delta_w, \delta_r)$, a saturation schedule noted $\widehat{\sigma} \in \Sigma(G)$ is defined by:*

$$\widehat{\sigma} \text{ is a saturation schedule } \iff RN_\sigma(G) = RS(G)$$

We note $\widehat{\Sigma}(G)$ the set of all saturation schedules.

In this section, we study how to compute $RS(G)$ and how to construct some saturation schedules in $\widehat{\Sigma}(G)$ for any DAG $G = (V, E, \delta, \delta_w, \delta_r)$.

### 3.2.2 Computing Register Need

Given a schedule $\sigma \in \Sigma(G)$, computing $RN_\sigma(G)$ is obvious and has a polynomial complexity $O(\overline{\sigma} \times |V_R|)$. In presence of a schedule, all values lifetime intervals are defined. Algorithm 1 computes the register need of a schedule.

**Algorithm 1** Computing register need

**Require:** the lifetime interval $L_u^\sigma$ of each value $u$
  $RN = 0$
  **for** $0 \leq i < \overline{\sigma}$ **do** {initialization}
    $vsa(i) = 0$
  **end for**
  **for all** $u \in V_R$ **do**
    **for** $i = 1 + \sigma(u) + \delta_w(u),\ kill_\sigma(u)$ **do**
      $vsa(i)++$
    **end for**
  **end for**
  **for** $0 \leq i < \overline{\sigma}$ **do** {searching the maximum vsa}
    **if** $vsa(i) > RN$ **then**
      RN=vsa(i)
    **end if**
  **end for**

### 3.2.3 Computing Register Saturation

When approaching the problem of computing register saturation, a first upper-bound of the register saturation is the amount of values produced within the DAG, i.e. $RS(G) \leq |V_R|$. Since some values could be killed by other operations, the killed value could not be alive simultaneously with its killing operations and its descendants. In this section, we give the formal problem formulation for computing an exact $RS(G)$.

When we have no schedule, value lifetime intervals are not defined. In this section, we study how to use precedence constraints defined by the initial DAG to deduce a schedule that use the

maximum amount of registers. A value $u$ is defined $\delta_w(u)$ steps after the schedule of $u$, but we cannot deduce which operation kills it since we do not assume any schedule. In the following, we will see that the register saturation problem is reduced to select a killing operation for each value.

Let us begin by assuming that we have killing function which associates to each value $u \in V_R$ one and only one killing operation $k(u) \in pkill(u)$.

**Definition 3.14 (Killing Function)** *Given a DDG $G = (V, E, \delta, \delta_w, \delta_r)$, a killing function $k$ is defined by*

$$\begin{array}{rccl} k: & V_R & \rightarrow & pkill(u) \\ & u & \mapsto & k(u) \end{array}$$

We use killing functions to chose an operation $k(u)$ to be the only one killer of a value $u \in V_R$, i.e. to satisfy the following assertion:

$$\forall u \in V_R,\ \exists v \in pkill(u) \quad \sigma(v) + \delta_r(v) = kill_\sigma(u) \Longleftrightarrow v = k(u) \tag{3.3}$$

This equation has to be verified for a family of schedules that we make precise below.

### Schedules Associated to a Killing Function

There is a family of schedules $\sigma \in \Sigma(G)$ that ensures the killing function assertion (3.3). Defining these schedules is obvious. For this purpose, we build an extended DAG $G\backslash^{E_k}$ such that all the schedules of this new DAG ensure the killing function assertion.

**Definition 3.15 (DAG Associated to a Killing Function)** *Given a DAG $G = (V, E, \delta, \delta_w, \delta_r)$ and a killing function $k$, the extended DAG associated to $k$ noted $G_{\rightarrow k} = G\backslash^{E_k}$ is defined by:*

$$E_k = \Big\{ e = (v, k(u))/u \in V_R : v \in pkill(u) - \{k(u)\} \wedge \delta(e) = \delta_r(v) - \delta_r\big(k(u)\big) + 1 \Big\}$$

That is we add serial arcs from all the potential killing operation $pkill(u)$ to the chosen one and only one killing operation $k(u)$. Then,

$$\forall \sigma \in \Sigma(G_{\rightarrow k})\ \forall u \in V_R\ \forall v \in pkill(u) - \{k(u)\} \quad \sigma\big(k(u)\big) + \delta_r\big(k(u)\big) > \sigma(v) + \delta_r(v)$$

We cannot choose an arbitrary killing function, because sometimes we could not be able to find a valid schedule that ensures the killing assertion (3.3). We must be sure that there is one valid schedule associated to it. So we give the condition for the validity of killing function.

**Definition 3.16 (Valid Killing Function)** *Given a DAG $G = (V, E, \delta, \delta_w, \delta_r)$ and a killing function $k$, then*

$$\begin{array}{rl} k \text{ is valid} & \Longleftrightarrow G_{\rightarrow k} \text{ is acyclic} \\ & \Longrightarrow \Sigma(G_{\rightarrow k}) \neq \phi \end{array}$$

Testing if a killing function $k$ is valid has a linear complexity $O(|V| + |E| + |E_k|)$. See appendix C for more details.

**Example 3.2.2** *Figure 3.6 describes an example where an arbitrary choice of killing operations is not correct. The original DAG is presented in part (a). For simplicity, all nodes are value and all edges are flow; the read delay is null for all operations. If we choose the killing function $k$ as described with the bold arcs (the source of the bold arc is killed by its destination), then this killing function is not valid since it introduces a cycle in the extended graph associated to $k$, see part (b).*

At this point, we can build a DAG that models values that can never be simultaneously alive. We first introduce the notion of descendant values.

**Definition 3.17 (Descendant Values)** *Given a DAG $G = (V, E, \delta, \delta_w, \delta_r)$, the descendant values set $\downarrow_{val} u$ of a node $u \in V$ is defined by:*

$$\downarrow_{val} u = \downarrow u \cap V_R$$

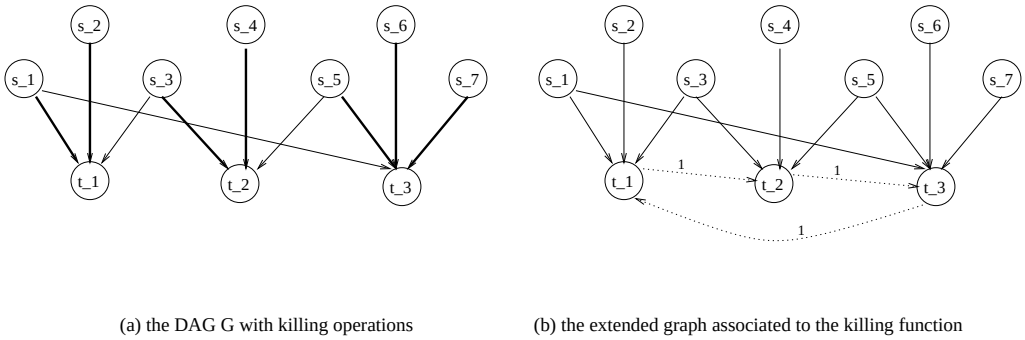
Then, any value $u$ can never be simultaneously alive with the $k(u)$'s descendant values. This is because, for any schedule that ensures the killing function $k$, the killing date scheduled with $k(u)$ is always before the definition of any descendant value in $\downarrow_{val} u$.

**Definition 3.18 (Disjoint Value DAG)** *Given a DAG $G = (V, E, \delta, \delta_w, \delta_r)$ and a killing function $k$, the disjoint value DAG of $G$, noted $DV_k(G) = (V_R, E_{DV})$ is defined by:*

$$E_{DV} = \left\{(u, v)/u, v \in V_R \ \wedge \ v \in \downarrow_{val} k(u)\right\}$$

Having a killing function $k$, the disjoint value DAG $DV_k(G)$ models the values that can never be simultaneously alive in any schedule that ensures $k$. The set of values that could be scheduled as simultaneously alive are all parallel values in $DV_k(G)$.

**Example 3.2.3** *Figure 3.7 gives an example to show how to build $DV_k(G)$ for a killing function $k$. The values $V_R$ and flow arcs are represented by bold lines in the DAG part (a). Part (b) gives the potential killing graph where a valid killing function $k$ is represented by bold arcs: the source of a bold arc is killed by its destination. The DAG $G_{\rightarrow k}$ associated to $k$ is presented in part (d): here we assume that reading from registers is done at cycle 0. Introduced serial arcs (see dashed ones) have then a null latency to ensure the killing function. Part (c) shows the disjoint value DAG $DV_k(G)$. For instance, $k(a) = c$ and $\downarrow_{val} c = \{c, e, g, h\}$ means that lifetime interval of $a$ is always before lifetimes intervals of $\{c, e, g, h\}$ for any schedule $\sigma \in \Sigma(G_{\rightarrow k})$ An antichain with maximal size in is the subset $\{c, d, e, g\}$.*




(a) the DAG G with killing operations (b) the extended graph associated to the killing function

Figure 3.6: Non valid killing function

**Theorem 3.1** *Given a DAG $G = (V, E, \delta, \delta_w, \delta_r)$ and a valid killing function $k$ then :*

- $\forall \sigma \in \Sigma(G_{\to k}) : RN_\sigma(G) \leq |AM_k|$
- $\exists \sigma \in \Sigma(G_{\to k}) : RN_\sigma(G) = |AM_k|$

*with $AM_k$ an antichain of maximal size in the disjoint value DAG $DV_k(G)$.*

**Proof :**

The disjoint value DAG $DV_k(G)$ models the order between value lifetime in any schedule of $G_{\to k}$. $\forall \sigma \in \Sigma(G_{\to k}), \forall u, v \in V_R$ :

$$u < v \text{ in } DV_k(G) \Longleftrightarrow u < k(u) \leq v \text{ in } G_{\to k}$$

If $v = k(u)$, then $\sigma(u) + \delta_w(u) < \sigma(v) + \delta_r(v)$, because of true data dependence. Since $\delta_r(v) \leq \delta_w(v)$, then $\sigma(u) + \delta_w(u) < \sigma(v) + \delta_w(v)$. In the case where $v \neq k(u)$, any path from $k(u)$ to $v$ is a data dependence path with strictly positive integer latencies; we deduce that

$$\forall \sigma \in \Sigma(G_{\to k}) \quad \sigma\big(k(u)\big) + \delta_r\big(k(u)\big) \leq \sigma(v) + \delta_w(v)$$

That is $kill_\sigma(u) \leq \sigma(v) + \delta_w(v)$; we deduce that following assertion is correct :

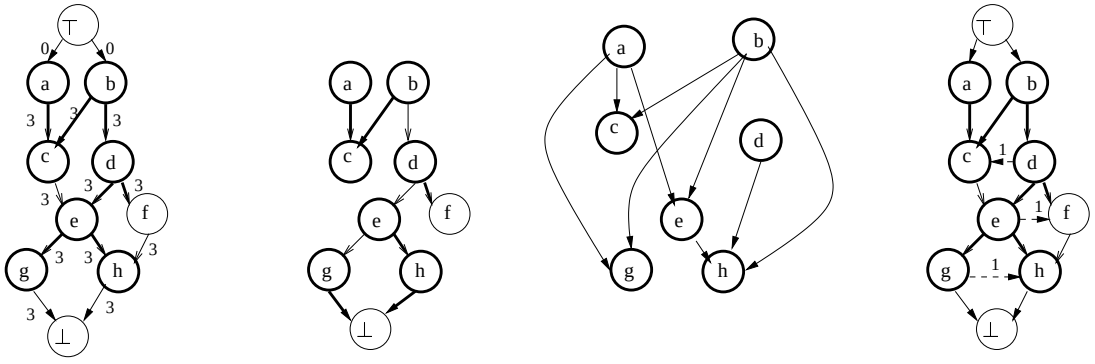
$$\forall \sigma \in \Sigma(G_{\to k}) \quad u \sim v \text{ in } DV_k(G) \implies L^\sigma_u \cap L^\sigma_v = \phi$$

We rewrite it : $\forall \sigma \in \Sigma(G_{\to k})$

$$\begin{array}{rcl} L^\sigma_u \cap L^\sigma_v \neq \phi & \implies & u || v \text{ in } DV_k(G) \\ & \implies & \{u, v\} \in vsa_\sigma(i)\,, i \in L^\sigma_u \cap L^\sigma_v \end{array}$$

Then, any values simultaneously alive for $\sigma \in \Sigma(G_{\to k})$ belong to an antichain in $DV_k(G)$ :

$$\forall 0 \leq i < \overline{\sigma},\ \exists A \text{ an antichain of } DV_k(G) \quad vsa_\sigma(i) \subseteq A$$




(a) G (b) $PK(G)$ with killing function (c) $DV_k(G)$ (e) $G_{\to k}$

Figure 3.7: Disjoint value DAG

Since $RN_\sigma(G_{\rightarrow k}) = \max_{0\leq i\leq\overline{\sigma}} |vsa_\sigma(i)|$ and $\forall 0 \leq i \leq \overline{\sigma}: \quad |vsa_\sigma(i)| \leq |AM_k|$, we conclude that $RN_\sigma(G) = \max_{0\leq i\leq\overline{\sigma}} |vsa_\sigma(i)| \leq |AM_k|$

Now, we have to build a schedule $\sigma$ such that $RN_\sigma(G) = |AM_k|$. For this purpose, we take $G_{\rightarrow k}$ in order to ensure the killing relation, and we add some serial arcs to enforce the values in $AM_k$ to be simultaneously alive. This leads us to a new extended DAG $G' = G_{\rightarrow k}\backslash^{E'}$ and

$$\forall \sigma \in \Sigma(G') \; \forall u, v \in AM_k : \qquad L^\sigma_u \cap L^\sigma_v \neq \phi$$

The necessary and sufficient condition which two values $u, v$ in $AM_k$ must satisfy to be simultaneously alive for any schedule of $G_{\rightarrow k}$ is

$$\begin{aligned} & v < u < k(v) \wedge lp(v,u) \geq \delta_w(v) - \delta_w(u) \; \wedge \\ & \qquad \wedge \; lp\big(u, k(v)\big) > \delta_w(u) - \delta_r\big(k(v)\big) && (3.4) \\ \vee \quad & u < v < k(u) \; \wedge \; lp(u,v) \geq \delta_w(u) - \delta_w(v) \wedge \\ & \qquad \wedge \; lp\big(v, k(u)\big) > \delta_w(v) - \delta_r\big(k(u)\big) && (3.5) \\ \vee \quad & k(u) = k(v) && (3.6) \end{aligned}$$

such that $lp(u,v)$ for $u, v \in V$ denotes the longest path from $u$ to $v$. These conditions ensure that $\forall \sigma \in \Sigma(G_{\rightarrow k}) \; \forall u, v \in V_R$:

$$\begin{aligned} u, v \text{ satisfy } (3.4) \quad & \Longrightarrow \quad \sigma(u) + \delta_w(u) \geq \sigma(v) + \delta_w(v) \wedge \\ & \qquad \wedge \, \sigma(k(v)) + \delta_r(k(v)) > \sigma(u) + \delta_w(u) \\ u, v \text{ satisfy } (3.5) \quad & \Longrightarrow \quad \sigma(v) + \delta_w(v) \geq \sigma(u) + \delta_w(u) \wedge \\ & \qquad \wedge \, \sigma(k(u)) + \delta_r(k(u)) > \sigma(v) + \delta_w(v) \\ u, v \text{ satisfy } (3.6) \quad & \Longrightarrow \quad kill_\sigma(u) = kill_\sigma(v) \end{aligned}$$

And then, by using interval order algebra defined in Sect. 2.2 (page 5):

$$\begin{aligned} u, v \text{ satisfy Cond. } (3.4) \quad & \Longrightarrow \quad \neg(L^\sigma_u \prec L^\sigma_v \; \vee \; L^\sigma_u \succ L^\sigma_v) \\ u, v \text{ satisfy Cond. } (3.5) \quad & \Longrightarrow \quad \neg(L^\sigma_u \succ L^\sigma_v \; \vee \; L^\sigma_u \prec L^\sigma_v) \\ u, v \text{ satisfy Cond. } (3.6) \quad & \Longrightarrow \quad L^\sigma_u \; f \; L^\sigma_v \end{aligned}$$

If two values in $u, v \in AM_k$ do not satisfy any of these conditions, then we use Algorithm 2 to enforce them. This algorithm use the boolean function $vsa_{G'}(u,v)$ to test if two values $u, v$ verify one of the above conditions. We add iteratively serial arcs until all values in $AM_k$ satisfy one of these conditions. The added serial arc does not introduce cycles and any schedule $\sigma$ of $G'$ has $RN_\sigma(G') = |AM_k|$. See Theorem 3.2.

⌟

**Theorem 3.2** *Let $G = (V, E, \delta, \delta_w, \delta_r)$ be a DAG and $k$ a killing function. The extended graph $G' = G_{\rightarrow k}\backslash^{E'}$ produced by Algorithm 2 is a DAG. And*

$$\forall u, v \in AM_k \; \forall \sigma \in \Sigma(G') \qquad L^\sigma_u \cap L^\sigma_v \neq \phi$$

*where $AM_k$ is a maximal antichain in $DV_k(G)$.*

**Algorithm 2** Extended $G_{\rightarrow k}$ to enforce values to be simultaneously alive

**Require:** a valid killing function $k$
construct the extended graph $G_{\rightarrow k}$ associated to $k$
$G' = G_{\rightarrow k}$ {the final extended graph is initialized}
search a maximal antichain $AM_k$ in the disjoint value DAG $DV_k(G)$
**for all** $u \in AM_k$ **do**
  **for all** $v \in AM_k / \ u \neq v$ **do**
    **if** $\neg vsa_{G'}(u,v)$ **then**
      **if** $u||v$ in $G'$ **then**
        **if** $\neg(k(u) < v)$ **then**
          add the serial arcs $e = (u,v), e' = (v,k(u))$ to $G'$ with $\delta(e) = \delta_w(u) - \delta_w(v)$ and $\delta(e') = \delta_w(v) - \delta_r\big(k(u)\big) + 1$
        **else** {$\neg(k(v) < u)$ certainly}
          add the serial arcs $e = (v,u), e' = (u,k(v))$ to $G'$ with $\delta(e) = \delta_w(v) - \delta_w(u)$ and $\delta(e') = \delta_w(u) - \delta_r\big(k(v)\big) + 1$
        **end if**
      **else**
        **if** $v < u$ **then**
          add the serial arcs $e = (v,u)$ and $e' = (u,k(v))$ to $G'$ with $\delta(e) = \delta_w(v) - \delta_w(u)$ and $\delta(e') = \delta_w(u) - \delta_r\big(k(v)\big) + 1$
        **else** $\{u < v\}$
          add the serial arcs $e = (u,v)$ and $e' = (v,k(u))$ to $G'$ with $\delta(e) = \delta_w(u) - \delta_w(v)$ and $\delta(e') = \delta_w(v) - \delta_r\big(k(u)\big) + 1$;
        **end if**
      **end if**
    **end if**
  **end for**
**end for**

**Proof :**

We proceed by induction. We prove that after exiting Algorithm 2, $G'$ is still a DAG. We prove also that the algorithm makes all values in $AM_k$ verifying one of the conditions Cond .(3.4), Cond. (3.5) or Cond. (3.6). For this last condition, if two values do not verify it in the DAG $G_{\rightarrow k}$, they cannot verify it in $G'$ : this is because the killing operations has been fixed in $G_{\rightarrow k}$. So, if $u, v$ do not verify Cond. (3.6), Algorithm 2 can only force them to verify Cond .(3.4) or Cond. (3.5).

We prove also the following property

$$\forall u, v \in AM_k \quad \neg\big(k(v) < u \vee k(u) < v\big) \text{ in } G'$$

which is the same as proving that algorithm 2 guarantees all values in $AM_k$ are forced to be simultaneously alive in $G'$ :

$$\nexists u, v \in AM_k / u \sim v \text{ in } DV_k(G')$$

Initially, this is correct because $u, v \in AM_k \Longrightarrow u \notin \downarrow_{val} k(v) \wedge v \notin \downarrow_{val} k(u)$. In this proof, we note $G'_i$ the graph built after exiting iteration $i$.

Suppose that after exiting iteration $i-1$, $G'_{i-1}$ is still a DAG and

$$\forall u, v \in AM_k \quad \neg\big(k(v) < u \ \vee \ k(u) < v\big) \text{ in } G'_{i-1}$$

Let $u_i$ and $v_i$ be the two chosen values at iteration $i$ which do not verify any of the conditions. Let us prove now that $G'_i$ is still a DAG and the two chosen values $u_i, v_i \in AM_k$ verify one of the conditions after exiting iteration $i$. And also, we prove that after exiting this iteration

$$\nexists w \in AM_k / k(u_i) < w \ \vee \ k(v_i) < w \text{ in } G'_i$$

Our algorithm introduces serial arcs in four cases :

1. $u_i || v_i$ in $G'_{i-1}$, then
   - if $\neg(k(u_i) < v_i)$, the two introduced arcs $e = (u_i, v_i), e' = (v_i, k(u_i))$ cannot introduce cycle, because $u_i < k(u_i)$ in $G'_{i-1}$, see Fig. 3.8(a). Now they are verifying Cond. (3.5). And also after introducing these arcs,

     $$\nexists w \in AM_k / k(u_i) < w \ \vee \ k(v_i) < w \text{ in } G'_i$$

     Suppose the contrary is true, i.e.

     $$\exists w \in AM_k / k(u_i) < w \ \vee \ k(v_i) < w \text{ in } G'_i$$

     If $k(u_i) < w$ in $G'_i \Longrightarrow k(u_i) < w$ in $G'_{i-1}$ because we have not introduced a serial arc from $k(u_i)$, which is impossible because of induction hypothesis.
     If $k(v_i) < w$ in $G'_i \Longrightarrow k(v_i) < w$ in $G'_{i-1}$ because we have not introduced a serial arc from $k(v_i)$, which is also impossible because of induction hypothesis ;

- else $\neg(k(v_i) < u_i)$ certainly, because

  $v_i < k(v_i) < u_i \wedge u_i < k(u_i) < v_i \Longrightarrow u_i < v_i \wedge v_i < u_i$ in $G'_{i-1}$ (impossible)

  Then the introduced arcs $e = (v_i, u_i), e' = (u_i, k(v_i))$ cannot introduce any cycle because $v_i < k(v_i)$ in $G'_{i-1}$, see Fig. 3.8(b). Now they are verifying Cond. (3.4). And also after introducing these arcs,

  $$\nexists w \in AM_k / k(u_i) < w \ \vee \ k(v_i) < w \text{ in } G'_i$$

  The proof is similar to the case above ;

2. if $v_i < u_i$ in $G'_{i-1}$, then by induction hypothesis $\neg(k(v_i) < u_i)$ in $G'_{i-1}$. The two introduced arcs $e = (v_i, u_i)$ and $e' = (u_i, k(v_i))$ cannot cause any cycle. Now they are verifying Cond. (3.4). And also after introducing these arcs,

   $$\nexists w \in AM_k / k(u_i) < w \ \vee \ k(v_i) < w \text{ in } G'_i$$

   The proof is similar to the case above ;

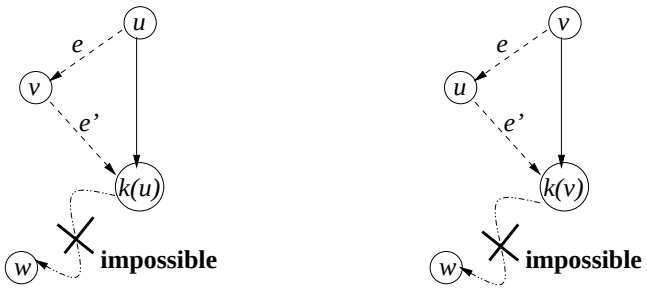
3. $u_i < v_i$ in $G'_{i-1}$, this case is similar to above. Now they are verifying Cond. (3.5).

After $n = |AM_k|^2$ iterations, we conclude that :

$$\forall u, v \in AM_k \quad u, v \text{ verify one of the conditions (3.4), (3.5) or (3.6)}$$

and then $\forall u, v \in AM_k \ \forall \sigma \in \Sigma(G') \qquad L^\sigma_u \cap L^\sigma_v \neq \phi$

⌟




(a) u < v <k(u) (b) v < u <k(v)

Figure 3.8: Forcing values to be simultaneously alive

For each valid killing function, we can deduce which values cannot be simultaneously alive with $u \in V_R$ in any schedule $\sigma \in \Sigma(G_{\rightarrow k})$. The following corollary defines them.

**Corollary 3.1** *Given a DAG $G = (V, E, \delta, \delta_w, \delta_r)$ and a valid killing function $k$, then $\forall u \in V_R$ :*

1. *the descendant values that cannot be simultaneously alive with u are the set of descendant values of the killing operation* $v \in \downarrow_{val} k(u)$ *:*

$$\forall \sigma \in \Sigma(G_{\rightarrow k}) \ \forall v \in \downarrow_{val} k(u) \qquad L^u_\sigma \prec L^v_\sigma \tag{3.7}$$

2. *the descendant values of u that could be simultaneously alive with u are those that do not belong to* $\downarrow_{val} k(u)$*, i.e. :*

$$\forall v \in \left( \bigcup_{v' \in pkill(u)} \downarrow_{val} v' \right) - \downarrow_{val} k(u) \ \exists \sigma \in \Sigma(G_{\rightarrow k}) \ : \ L^u_\sigma \cap L^v_\sigma \neq \phi \tag{3.8}$$

**Proof :**

Statement (3.7) is obvious since $v \in \downarrow_{val} k(u) \Longrightarrow u < v$ in $DV_k(G)$.

For Statement (3.8), we deduce it from Theorem 3.1. Since

$$\forall v \in \left( \bigcup_{v' \in pkill(u)} \downarrow_{val} v' \right) - \downarrow_{val} k(u) \quad u || v \text{ in } DV_k(G)$$

then $\exists A$ an antichain in $DV_k(G) / \ u, v \in A$. We build an extended DAG $G_A = G_{\rightarrow k} \backslash^{E'}$ to enforce all operations in $A$ to be simultaneously alive with $u$, i.e. we enforce one of the conditions Cond. (3.4), Cond. (3.5) or Cond. (3.6). We proceed in the same manner described in Algorithm 2. The added serial arcs ensure that

$$\forall \sigma \in \Sigma(G_A) \ \forall v \in A \qquad L^\sigma_u \cap L^\sigma_v \neq \phi$$

⌟

Theorem 3.1 allows us to rewrite the register saturation formula as

$$RS(G) = \max_{k \text{ is a valid killing function}} |AM_k| \text{ with } AM_k \text{ a maximal antichain in } DV_k(G)$$

We refer to the problem of finding such killing function as the *maximizing maximal antichain* problem.

**Definition 3.19 (Maximizing Maximal Antichain Problem (MMA))** *Given a DAG* $G = (V, E, \delta, \delta_w, \delta_r)$*, find a valid killing function k such that:*

$$\forall k' \text{ a valid killing function of } G : |AM_k| \geq |AM_{k'}|$$

*with* $AM_k$ *a maximal antichain in* $DV_k(G)$*, and* $AM_{k'}$ *a maximal antichain in* $DV_{k'}(G)$*.*

That is we should find a killing function maximizing maximal antichains in disjoint value DAGs. We call it a **saturating killing function**. We also define the inverse function $k^{-1}$ which denotes the set of values killed by a node $u$:

$$\forall u \in V \qquad k^{-1}(u) = \{v \in V_R / \ k(v) = u\}$$

**Theorem 3.3** *Given a DAG* $G = (V, E, \delta, \delta_w, \delta_r)$*, finding a saturating killing function is NP-complete.*

**Proof :**

We prove this theorem for a sub-family of DAGs $G = (V, E, \delta, \delta_w, \delta_r)$ composed of only one connected bipartite i.e. $V = S \cup T$, such that all nodes are values and all edges are flow ($V = V_R$ and $E = E_R$). $S$ are parent values and $T$ are children values that consume parent values (all edges go only from $S$ to $T$). In this case, $G = PK(G)$ and then the maximizing maximal antichain problem can easily be reduced to the minimum killing set problem (MKS is proven NP-complete) described in[BGS92]. To simplify writing this proof, we note the DAG as $G = (V = S \cup T, E)$, since the latencies are not used for finding a saturating killing function.

**Definition 3.20 (Minimum Killing Set (MKS))** *:*
*Given a bipartite DAG $G = (V = S \cup T, E)$, the minimum killing set of $G$ is a subset $T' \subseteq T$, such that*

1. *covering constraints*

$$\bigcup_{t \in T'} \Gamma_G^-(t) = S$$

2. *minimizing constraints*

$$\min |T'|$$

**Definition 3.21 (MMA decision problem: dec(MMA))** *Given a DAG $G = (V = S \cup T, E)$ which is a connected bipartite DAG and $V = V_R \wedge E = E_R$, and a strictly positive integer $j$, does there exist a valid killing function $k$ such that $|AM_k| \geq j$ ? where $AM_k$ is a maximal antichain in $DV_k(G)$.*

**Definition 3.22 (MKS decision problem: dec(MKS))** *Given a DAG $G = (V = S \cup T, E)$ which is a connected bipartite DAG with $V = V_R \wedge E = E_R$ and having an integer $j$, does there exist a minimum killing set $T' \in T$ such that $|T'| \leq j$ ?*

First, dec(MMA) belongs to NP. Having a killing function $k$, we can test in polynomial time ($O(|V| + |E|)$) if it is valid (see chapter C). Building the disjoint value DAG $DV_k(G)$ associated to $k$ can also be done in $O(|V| + |E|)$. Searching for a maximal antichain in $DV_k(G)$ is also polynomial thanks to Dilworth's decomposition. Second, we have to prove that dec(MMA) can be reduced to dec(MKS).

**dec(MKS) $\Longrightarrow$ dec(MMA)** Suppose we have a minimum killing set $T'$. We construct a valid killing function thanks to algorithm 3.

Now we have to prove two assertions :

1. this algorithm ensures that $k$ is a valid killing function ;
2. $k$ is a saturating killing function.

**Algorithm 3** Computing saturating killing function $k$

**Require:** a connected two-bipartite DAG $G = (S \cup T, E)$ and a minimum killing set $T'$
  initialize $k(s) = \perp$ for all parents $s \in S$
  **for all** nodes $t \in T'$ **do**
    **for all** parent $s \in \Gamma_G^-(t)$ **do**
      **if** $k(s) \neq \perp$ **then**
        put $k(s) = t$
      **end if**
    **end for**
  **end for**

For proving the first assertion, we need a new formulation of the potential kill relation between the nodes of $S$ and $T$. For the DAG component $G = (S \cup T, E)$, we define an hypergraph $\mathcal{H} = (S, \mathcal{E})$ with $\mathcal{E} = \{E_{t_0}, E_{t_1}, ..., E_{t_m}\}$ such that $E_{t_i} = \{s \in S / (s, t_i) \in E\}$[5]. This hypergraph models the relation between potential killing sets. Values belonging to the same edge $E_t \in \mathcal{E}$ could all potentially be killed by $t$. Figure 3.9 shows the hypergraph constructed for the example 3.2.2.

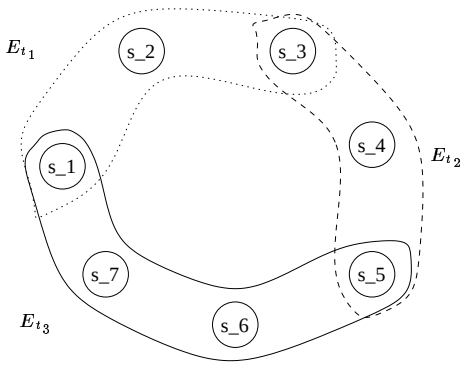



Figure 3.9: Hypergraph associated to killing function

We begin by proving that choosing a non valid killing function arises only if the hypergraph $\mathcal{H}$ is cyclic. Suppose that the killing function produced is non valid. This means that there is a cycle in $G_{\rightarrow k} = G \backslash^{E_k} = (V, E_{\rightarrow k})$. Since $G$ is bipartite (there is no arc from $T$ to $S$), and since in $G_{\rightarrow k}$ we add only arcs within $T$ nodes, so no arc can be introduced from $T$ to $S$. The only cycle that can be produced in $G_{\rightarrow k}$ is $C = (t_0, \ldots, t_n, t_0)$ where $t_i \in T$. We know by definition of a bipartite DAG that $\forall t_0, t_1 \in T : \ t_0 || t_1$ in $G$. Then

$$(t_0, t_1) \in E_{\rightarrow k} \Longrightarrow (t_0, t_1) \in E_k$$

[5] by definition, $t_i \in T$

In the hypergraph $\mathcal{H}$ we have :

$$\begin{array}{rcl}
(t_0,t_1)\in E_k & \Longrightarrow & \exists s\in S \qquad t_0\in pkill(s)\wedge t_1\in pkill(s)\wedge t_1=k(s)\\
 & \Longrightarrow & E_{t_0}\cap E_{t_1}\neq\phi\\
\vdots & & \\
(t_i,t_{i+1})\in E_k & \Longrightarrow & E_{t_i}\cap E_{t_{i+1}}\neq\phi\\
\vdots & & \\
(t_n,t_0)\in E_k & \Longrightarrow & E_{t_n}\cap E_{t_0}\neq\phi
\end{array}$$

Then

$$C'=(E_0,\ldots,E_n,E_0) \text{ is a cycle in } \mathcal{H} \tag{3.9}$$

The algorithm described above proceeds by choosing children $t_i\in T$, i.e. $E_{t_i}\in\mathcal{E}$ in a certain order. The properties induced by the heuristic are :

1. Each edge $E_{t_i}$ is selected at most once. So there is a visiting order "$<$" between hypergraph edges. We write $E_{t_i}<E_{t_j}$ if $E_{t_i}$ is selected before $E_{t_j}$.
2. If an edge $E_{t_i}$ is visited before $E_{t_j}$, then the common parents $(E_{t_i}\cap E_{t_j})$ are killed by $t_i$ or by another child visited before $E_{t_i}$. In any case, there are no common parents killed by $t_j$. Formally, we write

$$E_{t_i}<E_{t_j}\iff\forall s\in E_{t_i}\cap E_{t_j} : k(s)=t_i\vee k(s)\neq t_j \tag{3.10}$$

Now, we come back to our cycle $C$. If such a cycle exists, we have from (3.9) and (3.10) :

$$\begin{array}{rcl}
(t_i,t_{i+1})\in E_k & \Longrightarrow & \forall s\in E_{t_i}\cap E_{t_{i+1}} : k(s)=t_i\vee k(s)\neq t_{i+1}\\
 & & \text{otherwise } (t_i,t_{i+1})\notin E_{\rightarrow k}\\
 & \Longrightarrow & E_{t_i}<E_{t_{i+1}}
\end{array}$$

In the cycle $C'$, we have : $E_0<E_n<E_{n-1}<\ldots<E_1<E_0$. Contradiction !

Now, we know that $k$ is valid, and let us prove that it is a saturating one i.e. :

$$\forall \text{ valid killing function } k' \qquad |AM_{k'}|\leq|AM_k|$$

where $AM_k$ and $AM_{k'}$ denote maximal antichains in $DV_k(G)$ and $DV_{k'}(G)$. Having this valid killing function $k$ in $G$, $|AM_k|=|S|+|T''|$ is the amount of all parents values $S$ (because they are parallel by definition) and some children values $T''$. We prove that $|T''|=|T-T'|$ where $T'$ is the minimum killing set. By definition on $T'$, each parent $s$ in $S$ has at least one child in $T'$ that kills it. In $DV_k(G)$ we have,

$$\forall s\in S\ \exists t\in T' : k(s)=t\Longrightarrow\exists(s,t)\in E_{DV}$$

This means that only the parent or the child killer can belong to $AM_k$ Since $AM_k$ is a maximal antichain, then the following is true :

1. $\forall t \in T'$, if $|k^{-1}(t) = \{s\}| = 1$ then

$$s \in AM_k \otimes t \in AM_k$$

where $\otimes$ denotes the XOR operator. This is true because there is one and only one arc adjacent to $s$ and $t$ in $DV_k(G)$.

2. $\forall t \in T'$, if $|k^{-1}(t)| > 1$ then

$$k^{-1}(t) \subseteq AM_k \ \wedge \ t \notin AM_k$$

otherwise $AM_k$ is not maximal.

3. $\forall t \in T - T' \qquad t \in AM_k$, because $t$ is not connected to any value node in $DV_k(G)$.

Then, $AM_k = X_1 \cup X_2 \cup (T - T')$ is composed of three disjoint sets of values, see figure 3.10, where:

- the set of values $s \in S$ or $t \in T$ such that $|k^{-1}(t) = \{s\}| = 1$: $X_1 = \{v \in V / \exists t \in T' \ k^{-1}(t) = \{s\} \qquad (v = s) \otimes (v = t)\}$
- the set of parent values that have a common killer child: $X_2 = \{s \in S / k(s) = t \ \wedge \ \exists s' \in S - \{s\} \quad k(s') = t\}$

It is easy to see that $|X_1 \cup X_2| = |X_1| + |X_2| = |S|$. And then $|AM_k| = |S| + |T - T'| = |S| + |T''|$. Since $|S|$ is constant, searching for a saturating killing function is done by searching to maximize the set $|T''|$ and then to minimize $|T'|$. If $T'$ is the minimum killing set, then $k$ is a saturating killing function.

**dec(MMA) $\Longrightarrow$ dec(MKS)** Suppose we have a saturating killing function $k$ and let us prove that

$$T' = \bigcup_{s \in S} k(s)$$

is a minimum killing set. Suppose the contrary is true:

$$\exists T'' \subseteq T \qquad T'' \text{ is a minimum killing set} \wedge |T''| < |T'|$$

According to the above paragraph, $|AM_k| = |S| + |T - T'|$. If $T''$ exists, that means we can choose another valid killing function $k'$ thanks to algorithm 3 with $|AM_{k'}| = |S| + |T - T''|$. And we get the contradiction:

$$|T|' > |T''| \Longrightarrow |AM_{k'}| > |AM_k| \Longrightarrow k \text{ is not a saturating function}$$

⌟

### A Heuristic for MMA Problem

We present now our heuristic for approximating the MMA solution by a killing function $k^*$. We have to choose a killing operation for each value node such that we maximize the parallel values in the disjoint value DAG. Thanks to Prop. 3.3, we restrict our choice to only potential killing operations. We use a level per level approach, starting from the source nodes to the sinks in $PK(G)$. Our aim is to select a group of killing operations for a group of parents to keep as many descendant values alive as possible. The steps of our heuristic are :

1. decompose the potential killing DAG $PK(G)$ into connected bipartite components ;
2. for each bipartite component, search the best saturating killing set (defined later) ;
3. choose a killing operation within the saturating killing set.

Each step is explained in the following.

#### Decomposing $PK(G)$ into connected bipartite components :

We decompose the potential killing DAG into connected bipartite components (cbc) in order to choose a common saturating killing set for a group of parents. Our purpose is to have the maximum number of children and their descendants values simultaneously alive with their parents.

**Definition 3.23 (Connected Bipartite Component)** *Given a DAG $G = (V, E, \delta, \delta_w, \delta_r)$, a connected bipartite component $cb = (S_{cb}, T_{cb}, E_{cb})$ in the potential killing DAG $PK(G) = (V, E_{PK})$ is defined by :*

- *$E_{cb} \subseteq E_{PK}$ arcs are potential killing relations ;*
- *$cb = (S_{cb}, T_{cb}, E_{cb})$ is connected :*

$$\forall e_1, e_n \in E_{cb} \quad \exists \text{ a list } (e_1, \ldots, e_n) \ : \ e_i, e_{i+1} \text{ are adjacent, with } i = 1, \ldots n-1$$

- *any arc $e$ adjacent to an arc $e' \in E_{cb}$ also belongs to $E_{cb}$ :*

$$e \in E_{PK} \ \exists e' \in E_{cb} / \ e, e' \text{ are adjacent} \Longrightarrow e \in E_{cb}$$

- *$S_{cb} = \{s \in V_R / \ \exists e \in E_{cb} \wedge s = source(e)\}$ parent values ;*
- *$T_{cb} = \{t \in V / \ \exists e \in E_{cb} \wedge s = target(e)\}$ children nodes ;*

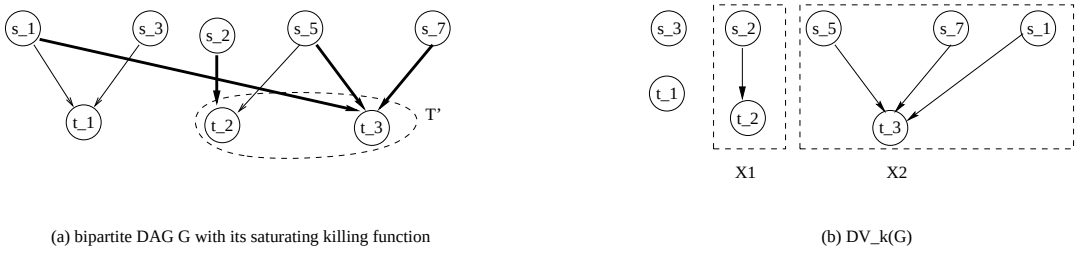



Figure 3.10: Minimum killing set and saturating function

- $cb = (S_{cb}, T_{cb}, E_{cb})$ *is bipartite :*

$$\nexists e, e' \in E_{cb} \quad target(e) = source(e')$$

**Definition 3.24 (Bipartite Decomposition)** *Given a DAG $G = (V, E, \delta, \delta_w, \delta_r)$, a bipartite decomposition of its potential killing DAG $PK(G)$ is the set*

$$\mathcal{B}(G) = \{cb = (S_{cb}, T_{cb}, E_{cb}) / \ \forall e \in E_{PK} \qquad \exists cb \in \mathcal{B}(G) : \ e \in E_{cb}\}$$

**Theorem 3.4** *Given a DAG $G = (V, E, \delta, \delta_w, \delta_r)$, then there exists only one bipartite decomposition $\mathcal{B}(G)$.*

**Proof :**

It is clear that $E_{PK} = \phi \Longrightarrow \mathcal{B}(G) = \phi$. Let us study the case where $E_{PK} \neq \phi$. Let us assume that there are two different bipartite decompositions $\mathcal{B}_1(G)$ and $\mathcal{B}_2(G)$ :

$$\exists cb = (S_{cb}, T_{cb}, E_{cb}) : \ cb \in \mathcal{B}_1(G) \wedge cb \notin \mathcal{B}_2(G)$$

Since a connected bipartite component $cb$ is defined by its arcs[6] set $E_{cb}$, this means that $\forall cb' \in \mathcal{B}_2(G)$ :

$$E_{cb} \neq E_{cb'} \Longleftrightarrow \forall E_{cb'}/cb' \in \mathcal{B}_2(G) \ \exists e \in E_{PK} \ e \in E_{cb} \wedge e \notin E_{cb'} \ \vee \ e \in E_{cb'} \wedge e \notin E_{cb}$$

In this proof, we study only the case where $e \in E_{PK} : \ e \in E_{cb} \wedge e \notin E_{cb'}$, because the proof for the second case is the same.

At this point, we make the difference between the two following cases :

1. $\exists E_{cb'} \quad E_{cb} \cap E_{cb'} \neq \phi$
2. $\nexists E_{cb'} \quad E_{cb} \cap E_{cb'} \neq \phi$

If $\exists E_{cb'} / \ E_{cb} \cap E_{cb'} \neq \phi$, let $e'$ be an arc from this intersection. For the purpose of this proof, we say that a list of adjacent arcs is adjacent if each arc is adjacent to its successor in this list. By definition, $E_{cb}$ and $E_{cb'}$ are connected. Since

$$e \in E_{cb} \Longrightarrow \exists \text{ a list in } E_{cb} \text{ of adjacent arcs } (e, \ldots, e') \tag{3.11}$$

and since $E_{cb'}$ must contain all other adjacent arcs until $e'$, we then have for any arc $e'' \in E_{cb'}$ :

$$\exists \text{ a list in } E_{cb'} \text{ of adjacent arcs } (e', \ldots, e'') \tag{3.12}$$

From (3.11) and (3.12) :

$$\exists \text{ a list of adjacent arcs } (e, \ldots, e'')$$

From bipartite component definition : $e \in E_{cb'}$ contradiction !

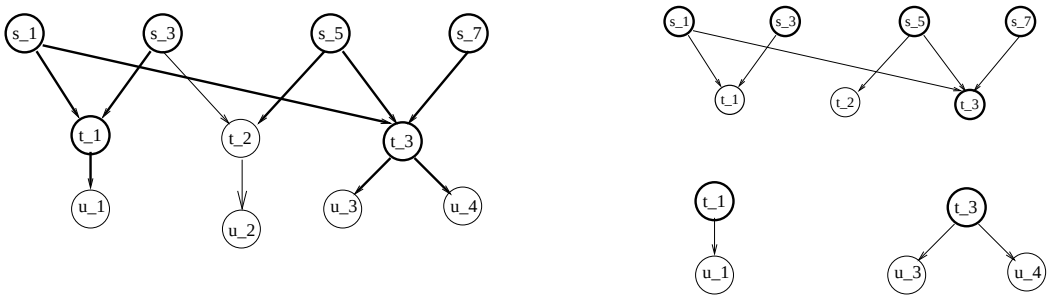
If $\nexists E_{cb'} / \ E_{cb} \cap E_{cb'} \neq \phi$, then :

[6] $S$ and $T$ are only sources and targets resp. of $E_{cb}$

1. either no $E_{cb}$ arc belongs to any bipartite component in $\mathcal{B}_2(G)$. Then $\mathcal{B}_2(G)$ is not a bipartite decomposition, since $E_{cb}$ arcs are uncovered ;
2. or no $E_{cb'}$ arc belongs to any bipartite component in $\mathcal{B}_1(G)$. Then $\mathcal{B}_1(G)$ is not a bipartite decomposition.

⌟

Appendix D gives our algorithm that computes the bipartite decomposition of a DAG in $O(|V| + |E_{PK(G)}|)$ complexity.




(a) the DAG

(b) bipartie components of the potential killling graph

Figure 3.11: Bipartite components

**Example 3.2.4** *Figure 3.11 shows an example of building bipartite components of the potential killing DAG. The DAG presented in part (a) has three bipartite components. Bold circles refer to value nodes, and bold arcs refer to flow arcs. Now, we should search for a killing function among children in each bipartite component saturating the register need.*

**Finding a saturating killing set :**

It is a subset $T'_{cb} \subseteq T_{cb}$ in the bipartite components $cb = (S_{cb}, T_{cb}, E_{cb})$ such that if we choose a killing operation in this subset, then we get a maximal number of descendant values simultaneously alive with values in $S_{cb}$. This choice maximizes the maximal antichain in the disjoint value DAG.

**Definition 3.25 (Saturating Killing Set (SKS))** *:*
*Given a DAG $G = (V, E, \delta, \delta_w, \delta_r)$, the saturating killing set $SKS(cb)$ of a connected bipartite component $cb = (S_{cb}, T_{cb}, E_{cb})$ in the potential killing DAG $PK(G)$ is a subset $T'_{cb} \subseteq T_{cb}$, such that*

1. *killing constraints : all parents must be covered by at least one child*

$$\bigcup_{t \in T'_{cb}} \Gamma^-_{cb}(t) = S_{cb}$$

*2. minimizing the cost*

$$\min |\bigcup_{t \in T'_{cb}} \downarrow_{val} t|$$

A SKS is only the subset of nodes $T'_{cb} \subseteq T_{cb}$ that could potentially kill $S_{cb}$ values, with a minimal amount of descendant values.

**Theorem 3.5** *Given a DAG $G = (V, E, \delta, \delta_w, \delta_r)$ a bipartite component $cb \in \mathcal{B}(G)$, computing $SKS(cb)$ is NP-complete.*

**Proof:**

See appendix E

⌟

**A heuristic for finding a SKS** Intuitively, we should choose a subset of children in bipartite component that kill most of parents while minimizing the number of descendant values. For this purpose, we define a cost function $\rho$ that permits us to choose the best candidate child. Given a bipartite component $cb = (S_{cb}, T_{cb}, E_{cb})$ and a set $Y$ of (cumulated) descendant values and a set $X$ of non killed parents, the cost of a child $t \in T_{cb}$ is :

$$\rho_{X,Y}(t) = \begin{cases} \frac{|\Gamma^-_{cb}(t) \cap X|}{|\downarrow_{val} t \cup Y|} & \text{if } \downarrow_{val} t \cup Y \neq \phi \\ |\Gamma^-_{cb}(t) \cap X| & \text{otherwise} \end{cases}$$

The first case permits us to select the child that covers the most uncovered parents with the minimum descendant values. If there is no descendant value, then we choose the child that covers the most uncovered parents.

Algorithm 4 gives a modified greedy heuristic that searches for an approximation $SKS^*$ of a saturating killing set and computes the killing function $k^*$ in polynomial time.

**Theorem 3.6** *The heuristic Greedy-k always produces a valid killing function.*

**Proof:**

The proof of this theorem is the same as for algorithm 3 in the proof of theorem 3.3. Rather than having a minimum killing set, we have now a saturating killing one : both sets have the same covering property, so we are sure that there is at least one potential killer for each parent. Since greedy-$k$ algorithm proceeds by choosing killing children $t_i \in SKS(cb)'$ in the same manner, the properties induced by this heuristic are the same as algorithm 3.

⌟

**Algorithm 4** Greedy-$k$: a heuristic for computing saturating killing functions $k^*$

**Require:** a DAG $G = (V, E, \delta, \delta_w, \delta_r)$
  **for all** values $u \in V_R$ **do**
    $k^*(u) = \perp$ {all values are initially non killed}
  **end for**
  construct all connected bipartite components of the potential killing DAG $PK(G)$.
  **for all** bipartite component $cb = (S_{cb}, T_{cb}, E_{cb})$ **do**
    $X = S_{cb}$ {all parents are initially uncovered}
    $Y = \phi$ {initially, no initial cumulated descendant values}
    $SKS^*(cb) = \phi$
    **while** $X \neq \phi$ **do** {build the SKS for $cb$}
      select the child $t \in T_{cb}$ with the maximal cost $\rho_{X,Y}(t)$ {$t$ is a new member of the SKS}
      $SKS^*(cb) = SKS^*(cb) \cup \{t\}$
      $X = X - \Gamma^-_{cb}(t)$ {remove covered parents}
      $Y = Y \cup \downarrow_{val} t$ {update the cumulated descendent values}
    **end while**
    **for all** $t \in SKS^*(cb)$ **do** {in decreasing cost order}
      **for all** parent $s \in \Gamma^-_{cb}(t)$ **do**
        **if** $k^*(s) = \perp$ **then** {kill non killed parents of $t$}
          $k^*(s) = t$
        **end if**
      **end for**
    **end for**
  **end for**

Now we can give the steps to compute the register saturation :

1. apply Greedy-$k$ on $G$. The result is a valid killing function $k^*$ ;
2. construct the disjoint value DAG $DV_{k^*}(G)$ ;
3. find a maximal antichain $AM_{k^*}$ of $DV_{k^*}(G)$ ; $RS^*(G) = |AM_{k^*}|$

**Remark** If there is no arc in the potential killing DAG ($E_{PK} = \phi$) then

$$\mathcal{B}(G) = \phi \Longleftrightarrow \forall u \in V_R \qquad k(u) = \bot$$

**Example 3.2.5** *Figure 3.12 shows an example of constructing one extended graph saturating the register need by using Algorithm 2. For simplicity, all the nodes of the original DAG (part a) are values except the two virtual nops ($\top, \bot$). The reading delay from a register is 0, and writing to it is 2 for all nodes. Also, all arcs are flow except those going to $\bot$ and coming from $\top$. Bold arcs describe the killing relation $k^*$ between nodes (the source of a bold arc is killed by its destination). The extended graph associated $G_{\rightarrow k^*}$ is presented in part (c). Now, we see that a maximal antichain of the disjoint value DAG (part b) is $\{a, b, c, d, g\}$, i.e $RS(G) = 5$. To build saturating schedules, each couple of saturating values must satisfy one of the condition (3.4), (3.5)or (3.6) in $G_{\rightarrow k^*}$. The values $\{a, b, c\}$ have the same killing operation $k^*(a) = k^*(b) = k^*(c) = e$, so they verify Cond. (3.6). The node $d$ is between $a$ and $k^*(a) = e$ but not with the suitable longest paths. So we introduce an arc from $d$ to $k^*(a) = e$ with the latency $\delta_w(d) - \delta_r(e) + 1 = 2 - 0 + 1 = 3$. Also, $\{g\}$ does not verify any of the conditions with $\{a, b, c\}$. Since $a < k^*(g) = \bot$, we have only the choice of introducing a serial arc from $g$ to $k^*(a) = e$ with the latency $\delta_w(g) - \delta_r(e) + 1 = 2 - 0 + 1 = 3$. The new extended DAG is given in part (d). Now, we can easily see that all the values $\{a, b, c, d, g\}$ are values simultaneously alive since they verify the conditions. Any schedule of the extended graph in part (d) has a register need of 5.*

### 3.2.4 Properties for non Connected DAGs

If the DAG $G = (V, E, \delta, \delta_w, \delta_r)$ is composed of a family of disjoint sub-DAGs $G_1, \ldots, G_m$ such that $G_i (0 \leq i \leq m)$ is connected, then

1. the register saturation is the sum of register saturation of each sub-DAG :

$$RS(G) = \sum_{i=1}^{m} RS(G_i)$$

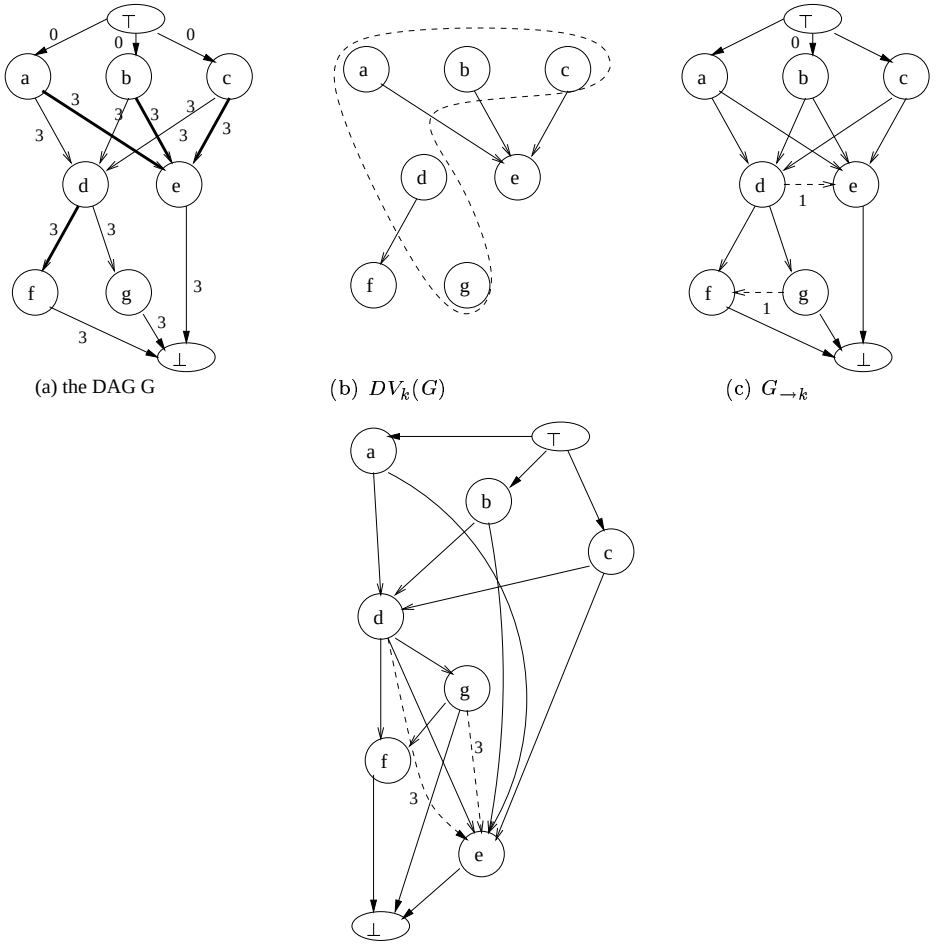
2. the saturating values are the union of saturating values of each sub-DAG :

$$AM = \bigcup_{0 \leq i \leq m} AM^i$$

   where $AM$ is the set of all saturating values and $AM^i$ is the set of saturating values of $G_i$.

3. the saturating values of each sub-DAG are disjoint :

$$\forall AM^i, AM^{i'},\ i \neq i' \qquad AM^i \cap AM^{i'} = \phi$$




(a) the DAG G

(b) $DV_k(G)$

(c) $G_{\rightarrow k}$

(d) an extended graph of $G_{\rightarrow k}$ that saturates the register use

Figure 3.12: Building saturating DAGs

## 3.3 Reducing Register Saturation

Given a DAG $G = (V, E, \delta, \delta_w, \delta_r)$, in this section we find an extended graph $\overline{G} = G\backslash^{\overline{E}}$ such that the register saturation is limited by a strictly positive integer (number of physical registers). Let $\mathcal{R}$ be this limit. Then :

$$\forall \sigma \in \Sigma(\overline{G}) : RN_\sigma(\overline{G}) \leq RS(\overline{G}) \leq \mathcal{R}$$

where $\overline{E}$ is the set of introduced serial arcs. If we succeed in reducing the register saturation, we can assure that any schedule could not use more than $\mathcal{R}$ registers. Then, any schedule and register allocation heuristic on $\overline{G}$ do not introduce spill code.

The concept of register sufficiency was studied in [AKR91]. We give formally its definition.

**Definition 3.26** *Given DAG $G = (V, E, \delta, \delta_w, \delta_r)$, the register sufficiency $RF(G)$ of $G$ is defined by*

$$RF(G) = \min_{\sigma \in \Sigma(G)} RN_\sigma(G)$$

It is clear that if $RF(G) > \mathcal{R}$, then spill code cannot be avoided. We do not treat spill code optimization in our work. But if $RF(G) \leq \mathcal{R}$, then we can modify $G$ by introducing some serialization arcs such that the register saturation of the new graph $RS(\overline{G}) \leq \mathcal{R}$.

Searching a schedule that minimizes the critical path with a limited number of physical registers is known to be NP-hard. In this section, we present a heuristic that adds serialization arcs to reduce the maximal antichain in the disjoint value DAG without increasing the critical path if possible.
Serializing two values $u, v \in V_R$ means to force them to never verify any of the conditions (3.4), (3.5) nor (3.6) defined in page 19. That is we must ensure that the killing date of $u$ should always be scheduled after $v$'s definition. This is done by adding serial arcs from all potential killing operations of $u$ to $v$. We apply these serializations iteratively within saturating values until we reduce the register saturation below the limit $\mathcal{R}$. The following definition formalizes this idea.

**Definition 3.27 (Value Serialization)** *Given a DAG $G = (V, E, \delta, \delta_w, \delta_r)$, a value serialization $u \rightarrow v$ for $u, v \in V_R$ is an extended graph of $G$ defined by :*

- *if $v \in pkill(u)$ then add the serial arcs*

$$\{e = (v', v)/v' \in pkill(u) - \{v\} \wedge \delta(e) = \delta_r(v') - \delta_w(v)\}$$

- *else add the serial arcs $\{e = (u', v)/u' \in pkill(u) \wedge \neg(v < u') \wedge \delta(e) = \delta_r(u') - \delta_w(v)\}$.*

**Proposition 3.4** *A value serialization applied to a DAG $G = (V, E, \delta, \delta_w, \delta_r)$ never introduces cycles*

**Proof :**

After applying a value serialization $u \rightarrow v$, we have two cases :

1. if $v \in pkill(u)$, then $\forall v' \in pkill(u) \quad v||v'$ (property of potential killing operation, page 12). Then the introduced arcs cannot produce cycles.
2. if $v \notin pkill(u)$, then the introduced arcs $\{(v', v)/v' \in pkill(u)\}$ verify the condition $\neg(v < v')$. So, no cycle can be produced.

⌟

Before searching for a value serialization, we should test if it is admissible, i.e. it can really serialize the two values lifetime intervals (no cycle prohibits it). We need this information to build in a further algorithm the set of all admissible value serializations in order to choose the best one.

**Definition 3.28 (Admissible Value Serialization)** *Given a DAG $G = (V, E, \delta)$, a value serialization $u \rightarrow v$ for $u, v \in V_R$ is admissible iff :*

$$\forall v' \in pkill(u) \qquad \neg(v < v')$$

**Example 3.3.1** *Figure 3.13 gives an example of value serialization. The value nodes of the DAG (part a) are in bold circles, and flow arcs are bold also. Write latency is 2, and read latency is 0. Part (b) gives the extended graph if we serialize $s_1 \rightarrow s_5$ : since $pkill(s_1) = \{t_1\}$, we add the serial arcs $(t_1, s_5)$ with latency 1. Part (c) shows the extended graph if we serialize $s_5 \rightarrow u_2$ : since $pkill(s_5) = \{t_2, t_3\}$, we add the serial arcs $(t_3, u_2)$ and $(t_2, u_2)$. Part (d) gives an example of non admissible value serialization $s_7 \rightarrow s_1$ since $pkill(s_7) = \{t_3\} \wedge s_1 < t_3$.*

Now we give the skeleton of our heuristic that uses value serialization to reduce register saturation :

1. use Greedy-$k$ to build a saturating killing function ;
2. construct the maximal antichain $AM_k$ of the disjoint value DAG ;
3. if $|AM_k| \leq \mathcal{R}$ then exit ;
4. construct the set $U_k$ of all admissible value serializations between values in $AM_k$ ;

$$U_k = \{(u, v)/u, v \in AM_k \ \wedge \ u \rightarrow v \text{ is admissible}\}$$

5. do a value serialization that maximizes a cost $\omega$ (defined later) ;
6. update the maximal antichain $AM_k$ in the disjoint value DAG ;
7. go to 1;

Algorithm 5 gives our heuristic. It iterates value serializations within saturating values until we get the limit $\mathcal{R}$. The heuristic fails if $RF(G) > \mathcal{R}$ since we cannot reduce more than $RF(G)$ values simultaneously alive . If the heuristic fails in reducing the register saturation under the limit $\mathcal{R}$, it produces a more limited register saturation. Since this new graph produces a number of values simultaneously alive lower than the original one, the spill code produced after by any schedule is also limited.

The cost function that we define further allows us to guide the choice of a value serialization that inhibits the maximum amount of saturating values to be simultaneously alive, without increasing the critical path. If it is not possible, we choose a value serialization that produces the minimal critical path. So we need two parameters $\omega(u \rightarrow v) = (\omega_1, \omega_2)$ with

- $\omega_1 = \mu_1 - \mu_2$ is the prediction of the reduction obtained in the maximal antichain if we do the serialization, where
  - $\mu_1$ is the amount of saturating values serialized in $AM_k$ if we do the serialization ;
  - $\mu_2$ is the expected number of values that become saturating ones if we do the value serialization ;
- $\omega_2$ is the increase in the critical path length if we do the serialization.

At the end of the algorithm, we apply a general verification to ensure the potential killing property proven in Prop. 3.3, as explained below.

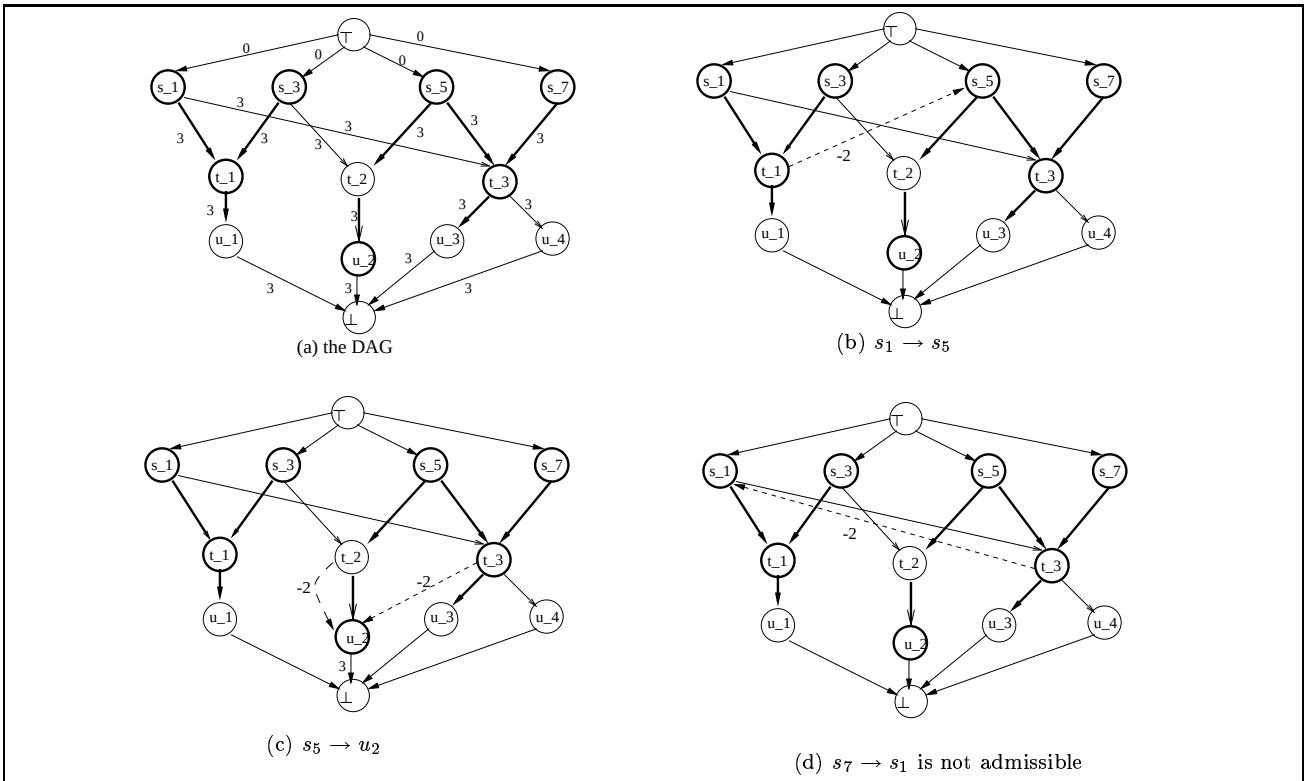



Figure 3.13: Value serialization

**Algorithm 5** Heuristic for reducing register saturation

**Require:** a DAG $G = (V, E, \delta, \delta_w, \delta_r)$ and a strictly positive integer $\mathcal{R}$
  $\overline{G} = G$
  construct $AM_k$ the saturating value nodes of $\overline{G}$;
  **while** $|AM_k| > \mathcal{R}$ **do**
    construct the set $U_k$ of all admissible serializations between saturating values in $AM_k$ with their costs $(\omega_1, \omega_2)$;
    **if** $\nexists (u \rightarrow v) \in U / \omega_1(u \rightarrow v) > 0$ **then** {no more possible reduction}
      exit;
    **end if**
    $X = \{(u \rightarrow v) \in U / \omega_2(u \rightarrow v) = 0\}$ {the set of value serializations that do not increase the critical path}
    **if** $X \neq \phi$ **then**
      choose a value serialization $(u \rightarrow v)$ in $X$ with the minimum cost $\mathcal{R} - \omega_1$;
    **else**
      choose a value serialization $(u \rightarrow v)$ in $X$ with the minimum cost $\omega_2$;
    **end if**
    do the value serialization $(u \rightarrow v)$ in $\overline{G}$;
    compute new saturating values $AM_k$ of $\overline{G}$;
  **end while**
  ensure potential killing operations property {check longest paths between pkill operations}

### Ensure Potential Killing Operations Property

The extended DAG produced at the end of the algorithm must not violate the potential killing operations property proven in Prop. 3.3 in page 13 for the initial DAG : we have proven that operations that do not belong to $pkill(u)$ cannot kill the value $u$. Otherwise, the register saturation computed for the final extended DAG would not be correct. We have then to ensure the following assertion :

$$\forall u \in V_R,\ \forall v, v' \in Cons(u) \quad v' < v \Longrightarrow lp(v', v) > \delta_r(v') - \delta_r(v) \tag{3.13}$$

This property is verified in the initial DAG because its arcs represent data dependencies with strictly positive latencies. If we introduce new arcs, we must also guarantee assertion (3.13). In fact, this problem occurs when we create a path from $v'$ to $v$ such that $v, v' \in pkill(u)$ for $u \in V_R$. If such a path has been created and $lp(v', v) \leq \delta_r(v') - \delta_r(v)$, then we can schedule $v'$ such that it kills the value $u$. Since our register saturation problem formulation assumes that $v'$ cannot kill $u$, we have to ensure it in the final extended DAG. This is done by adding a serial arc $e = (v', v)$ with $\delta(e) = \delta_r(v') - \delta_r(v) + 1$.

Figure 3.14 explains how we solve this problem when we do successive value serializations. On the left, we give a part of the initial DAG : all nodes are values and all arcs are flow. Read latency is 0 and the write is delayed by 2 cycles. Part (1) shows the extended DAG if we do the value serialization $b \rightarrow d$. Part (2) presents the second step where we do $a \rightarrow f$. At this point, we see that a path of length -6 has been introduced between $e$ and $f$ then $e \notin pkill(c)$. So, we assume that $e$ cannot kill $c$. However, since the longest path from $e$ to $f$ is -6, this violate assertion (3.13), because we can have a schedule such that $e$ kills $c$. In part (3), we introduce a serial arc with latency $\delta_r(e) - \delta_r(f) + 1 = 0 - 0 + 1 = 1$ to verify this assertion.

### Costs of Value Serializations

We explain here how to compute the parameters $\mu_1, \mu_2, \omega_2$ by using Cor. 3.1. We note $\overline{G_i}$ the extended DAG of step $i$, $k_i$ its saturating function, and $AM_{k_i}$ its saturating values. We note also $pkill_i(u)$ the set of potential killing operations in $\overline{G_i}$, and $\downarrow_{val_i} u$ the descendant values of $u$ in $\overline{G_i}$. The purpose of the cost function is to predict the reduction in register saturation introduced when we extend $\overline{G_i}$ to $\overline{G_{i+1}}$ with a value serialization $(u \rightarrow v)$.

1. if we do an admissible value serialization $(u \rightarrow v)$ in $\overline{G_i}$, then we ensure that the killing date of $u$ is always before $v$'s definition and all its descendants. Then, we reduce $AM_{k_i}$ by :
$$\mu_1 = |\downarrow_{val_i} v \cap AM_{k_i}|$$
which is the amount of saturating values in $\overline{G_i}$ that cannot be simultaneously alive with $u$ in $\overline{G_{i+1}}$;

2. if we do an admissible value serialization $(u \rightarrow v)$ in $\overline{G_i}$, then we could create new saturating values in $\overline{G_{i+1}}$ : this is because we could force its saturating killing function $k_{i+1}$ :
   - if $v \in pkill_i(u)$, then we force $k_{i+1}(u) = v$. The number (using Cor. 3.1)
   $$\mu_2 = \left| \left( \bigcup_{v' \in pkill_i(u)} \downarrow_{val_i} v' \right) - \downarrow_{val_i} v \right|$$
   is the amount of values in $\overline{G_i}$ that could be simultaneously alive with $u$ in $\overline{G_{i+1}}$.
   - else $\mu_2 = 0$ because $k_{i+1}(u)$ does not change, so we predict that we do not introduce new saturating values;

3. if we do an admissible value serialization $(u \rightarrow v)$ in $\overline{G_i}$, the introduced serial arcs could enlarge the critical path. Let $lp_i(v', v)$ be the longest longest path going from $v'$ to $v$ in $\overline{G_i}$[7]. If we introduced a serial arc $e = (v', v)$ with a latency $\delta(e)$, then the new longest path going from $\top$ to $\bot$ through $(v', v)$ in $\overline{G_{i+1}}$ is equal to :
$$lp_i(\top, v') + lp_i(v, \bot) + \delta(e) \text{ if } \delta(e) > lp_i(v', v)$$
Then, the new longest path in $\overline{G_{i+1}}$ through added serial arcs is
$$\max_{\substack{\text{introduced } e=(v',v) \\ \delta(e) > lp_i(v',v)}} lp_i(\top, v') + lp_i(v, \bot) + \delta(e)$$

[7]is such path does not exists, we assume $-\infty$

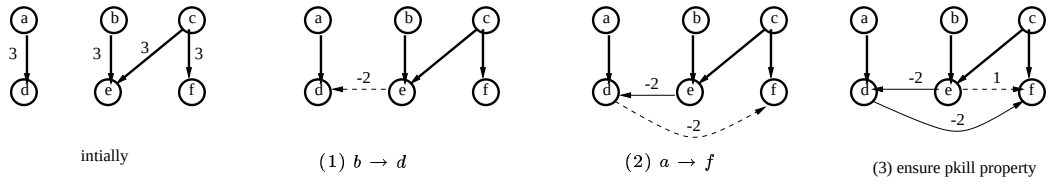



Figure 3.14: Ensure potential killing operations property

If this path is shorter than the critical path of $\overline{G_i}$, then $\omega_2 = 0$. Otherwise, $\omega_2$ is the difference between the original longest path and the new introduced longest path.

Thanks to our costs $\omega_1, \omega_2$, we can choose the most suitable value serialization :

1. choose the value serialization that does not increase the critical path ($\omega_2 = 0$) and maximize the reduction of values simultaneously alive : we choose a value serialization that minimizes $\mathcal{R} - \omega_1$ ;
2. if all value serializations increase the critical path, choose one that minimizes $\omega_2$.

**Example 3.3.2** *Figure 3.15 shows how our heuristic reduces the register saturation for the DAG of part (1). For simplicity, all nodes except $\perp$ and $\top$ are values, and all arcs are flow except those adjacent to $\perp$ or to $\top$. The read from physical register occurs at cycle 0 and write occurs in the last execution cycle ($\forall u \in V_R : \delta_w(u) = lat(u) - 1$). The saturating values of this DAG are $AM_k = \{a, b, c, d, g\}$[8], so $RS(G) = 5$. We want to reduce it to 4.*
*First, we construct the graph of all admissible value serializations (part 2) and their costs $(\omega_1, \omega_2)$. Let's to see how we compute them for $a \rightarrow d$ :*

- *if we do this serialization, we should add the arc $(e, d)$ in $G$ with latency -3. We will get in the extended DAG $\overline{G}$ that the saturating values $\downarrow_{val} d \cap AM_k = \{d, g\}$ cannot be simultaneously alive with $a$. Then $\mu_1 = 2$. But, since we have forced $d$ to kill $a$, the value $\downarrow_{val} d \cup \downarrow_{val} e - \downarrow_{val} d = \{e\}$ can be simultaneously alive with $a$ in $\overline{G}$. Then $\mu_2 = 1 \Longrightarrow \omega_1 = \mu_1 - \mu_2 = 1$ ;*
- *the critical path is $(\top, a, d, f, \perp) = 28$. If we introduce the arc $(e, d)$ with latency -2, the longest path from $\top$ to $\perp$ through $e, d$ is 26. So, the critical path does not change. Then $\omega_2 = 0$.*

*Let's now to look for the cost of $d \rightarrow g$ :*

- *if we do this value serialization, we should add the arc $(f, g)$ in $G$ with latency -2. We will get in the extended DAG $\overline{G}$ that the saturating value $\downarrow_{val} g \cap AM_k = \{g\}$ can be simultaneously alive with $d$ in $\overline{G}$. Then $\mu_1 = 1$. Since we force $g$ to kill $d$, the value $\downarrow_{val} g \cup \downarrow_{val} f - \downarrow_{val} g = \{f\}$ can be simultaneously alive with $d$ in $\overline{G}$. Then $\mu_2 = 1 \Longrightarrow \omega_1 = \mu_1 - \mu_2 = 0$ ;*
- *If we introduce the arc $(f, g)$ with latency -2, the longest path through $(f, g)$ is $(\top, a, d, f, g, \perp) = 14$. Since the critical path in $G$ does not increase, then $\omega_2 = 0$.*

*Now we have all admissible serializations. Our heuristic chooses $a \rightarrow d$ since it has a strictly positive $\omega_1$ and does not increase the critical path. The new extended graph is presented in figure 3.15.(3). At the end of the algorithm, we must ensure potential killing operations property described with assertion (3.13). After doing value serializations, we have created a path from $e$ to $d$ with a latency -2. Then, we can get a schedule such that $e$ kills $a$, even if $e \notin pkill_{\overline{G}}(a)$. To ensure that only operations from $pkill_{\overline{G}}(a) = \{d\}$ can kill $a$ , we add a serial arc $(e, d)$ with the latency $\delta_r(e) - \delta_r(d) + 1 = 0 - 0 + 1$. The final extended DAG $\overline{G}$ is then presented in part (4). The saturating values of this DAG are shown in part d. Now, we have the maximal antichain $AM_k = \{a, b, c, e\} \Longrightarrow RS(\overline{G}) = 4$.*

[8] similar to the one studied in example 3.2.5 in page 33

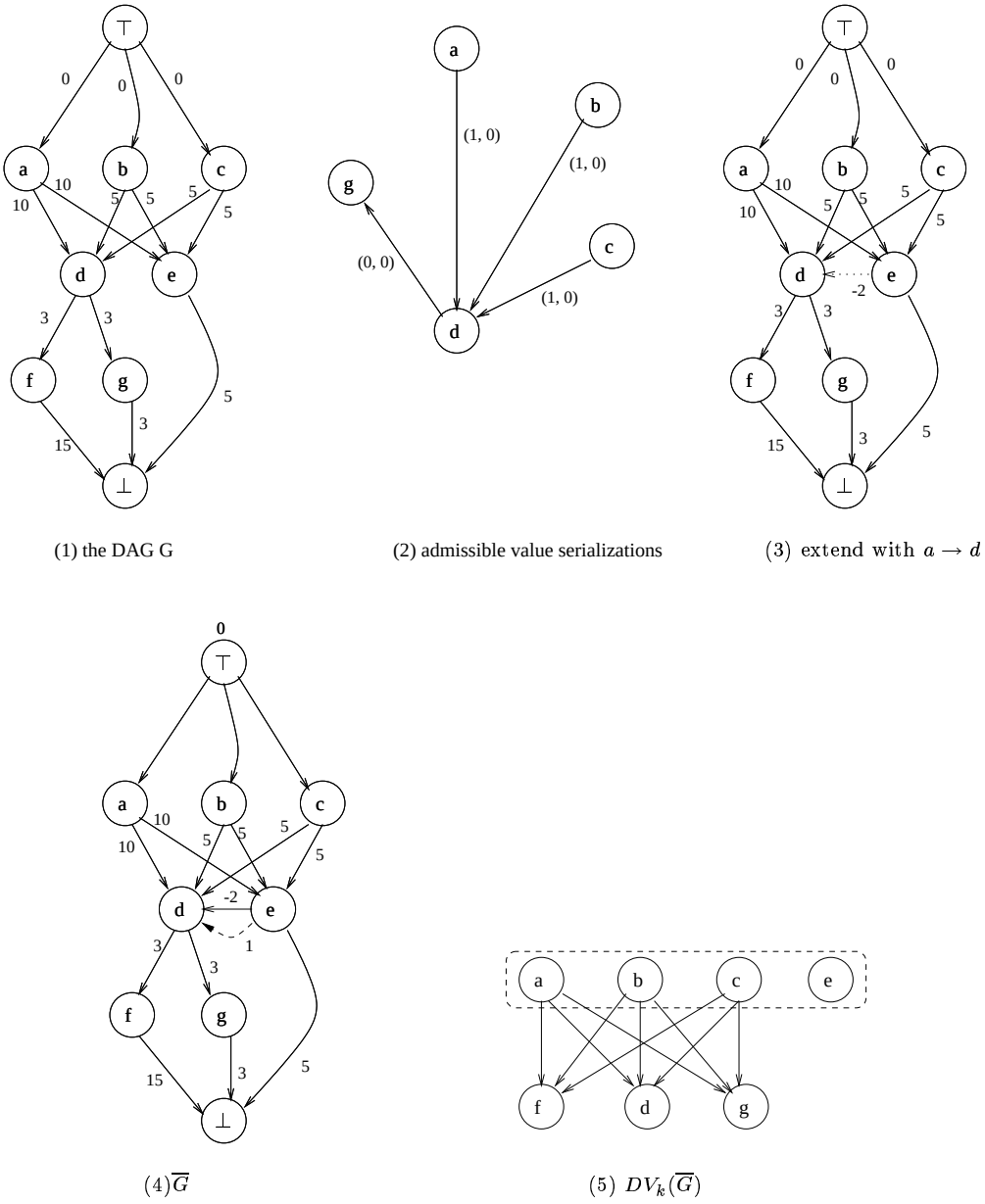



Figure 3.15: Reducing register saturation

## 3.4 Case of Forest of Inverted Trees

If the DAG $G = (V, E, \delta, \delta_w, \delta_r)$ representing precedence constraints is an inverted tree[9], then computing register saturation is optimal in polynomial complexity time.

We know that that the successors set of a node is singleton or a null set

$$G \text{ is an inverted tree } \iff \forall u \in V \quad |\Gamma^+_G(u)| \leq 1$$

Since each value has at most one consumer, it has only one possibility for the killing function. That is

$$\forall u \in V, \forall \sigma \in \Sigma(G) \quad kill_\sigma(u) = \sigma(v) \text{ where } pkill(u) = \{v\}$$

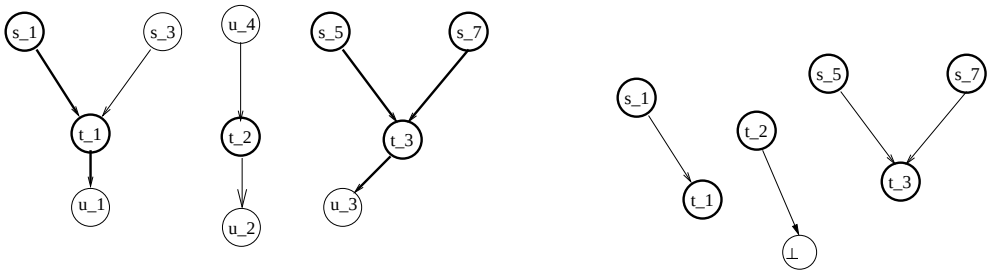
$$\Longrightarrow \{k(u)\} = \{pkill(u)\} \text{ is the optimal killing function}$$

Our heuristic gives this optimal result since it has only one choice for each bipartite component. The register saturation is then optimal and computed in polynomial time. Since the original graph is an inverted tree, then the disjoint value DAG must also be an inverted tree. The maximal antichain of the disjoint value DAG contains only its roots, i.e.

$$RS(G) = |Source\big(DV_k(G)\big)|$$

and the saturating values are $Source\big(DV_k(G)\big)$.

**Example 3.4.1** *See figure 3.16*




(a) $G$ is an inverted tree

(b) $DV_k(G)$ (each value have only one possible kill)

Figure 3.16: Disjoint value DAG is optimal with inverted trees

**Pure data flow trees** Rather than executing our algorithms in order to compute the register saturation, the special case where the tree represents a pure data flow graph permits us to give directly the register saturation without constructing the disjoint value DAG. For this purpose, $G = (V, E, \delta, \delta_w, \delta_r)$ must satisfy :

1. $G$ is an inverted tree ;
2. each node is a value node : $V_R = V$ ;

[9] each node has at most one child

3. each arc is a flow arc: $E_R = E$.

This sort of DAGs are for example those computing an arithmetic expression. Under these conditions, the disjoint value DAG is exactly the same tree after removing transitive arcs. Then,

$$RS(G) = |Source(G)|$$

and the saturating values are $Source(G)$.

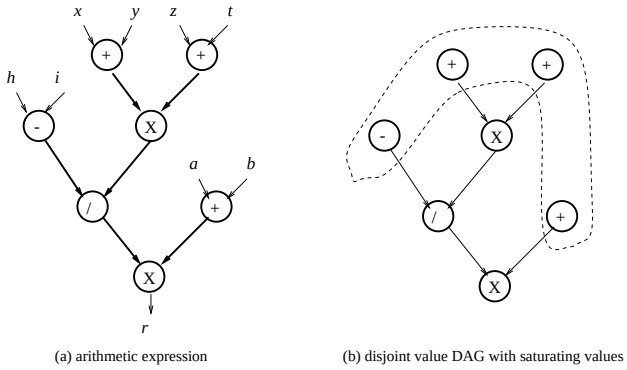
**Remarks** If this tree is an $n$-balanced tree[10] with a depth $d$, then

$$RS(G) = n^d$$

**Example 3.4.2** *Figure 3.17.a gives the data flow graph of the expression*

$$r = \frac{x \times y + z \times t}{h - i} \times (a + b)$$

*Since each value is killed by its child, and since all nodes are values (bold circles) and all edges are flow (bold arcs), then the disjoint value DAG $DV_k(G)$ is exactly the same tree, part (b). The saturating values are then the sources of $DV_k(G)$ which are also the sources of $G$. The register saturation is then 4 registers.*




(a) arithmetic expression (b) disjoint value DAG with saturating values

Figure 3.17: Pure data flow inverted trees

From above, we prove the following theorem.

**Theorem 3.7** *Given a DAG $G = (V, E, \delta, \delta_w, \delta_r)$, then*

$$PK(G) \text{ is an inverted tree} \implies \text{computing } RS(G) \text{ with Greedy-k is optimal}$$

[10] balanced tree such that intermediate nodes have exactly $n$ parents

**Proof :**

Since $PK(G)$ is an inverted tree, then $\forall u \in V_R \qquad |pkill(u)| = 1$. The optimal saturating killing function $k$ is defined by :

$$\forall u \in V_R \qquad k(u) = \{pkill(u)\}$$

This function is computed by our greedy-$k$ heuristic since in each bipartite component $cb = (S_{cb}, T_{cb}, E_{cb}) \in \mathcal{B}(G)$, the saturating killing set $SKS(cb)$ is composed of a single node $k(s) = t = \Gamma^-_{PK(G)}(s)$ for any $u \in S$ (since all parents have the same child). For any value $u \in V_R$ that is not read in $G$ this means that $pkill(u) = \{\bot\}$ and then $k(u) = \bot$. From theorem 3.1, we deduce that $RS(G) = |AM_k|$, with $AM_k$ a maximal antichain in $PK(G)$, is the optimal register saturation of $G$, with

$$AM_k = Source\big(PK(G)\big)$$

⌟

## 3.5 Case of Branches

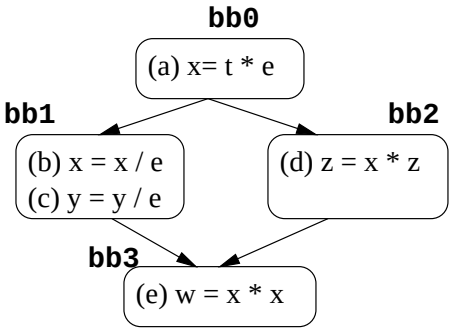
Our model assumes that there is only one possible definition per value. This assumption is correct if the code does not contain branches. If not, static data dependence analysis could provide for some values more than one definition because it cannot determine which execution path is taken.

On the other hand, the compiler can make some global scheduling to expose more ILP to the scheduler within each basic block (BB). Useful techniques like code motion, trace scheduling, hyperblock and superblock scheduling can be used to move operations across branch boundaries. Such static speculation could introduce new recovering operations to preserve the code semantic. These operations must be included in our DAG.

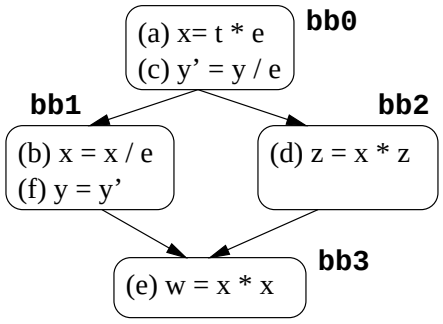
Our idea to handle branches is to take the code produced after a static speculation to get the new operations included in each BB and then build a DAG for each possible execution path in the control flow graph (CFG). At this point, we get only one possible definition per value in each DAG. We define $RSC(CFG)$ as the maximum register need that could be produced for any schedule of a speculated CFG :

$$RSC(CFG) = \max_{G \text{ build for each path in CFG}} RS(G)$$

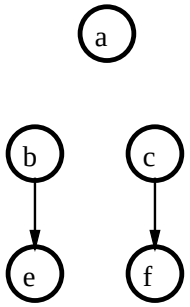
**Example 3.5.1** *Figure 3.18 shows how we handle branches. Part (1) gives the original CFG. The value x consumed in operation (e) could be defined by operation (a) or (b). A speculated CFG is shown in part (2) : operation (c) was moved to BB0 and its target destination renamed, while operation (f) is introduced in BB1. Parts (3) gives the DAG built for the path $(bb_0, bb_1, bb_3)$ : the value x consumed by operation (e) is now defined by operation (b). Part (4) gives the DAG built for the path $(bb_0, bb_2, bb_3)$ : the value x consumed by operation (e) is now defined by operation (a). In each path, we have only one possible definition per value.*



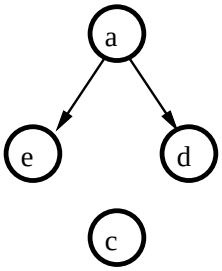

(1) Original CFG




(2) Speculated CFG




(3) DAG through the first path




(4) DAG through the second path

Figure 3.18: Register Saturation in Case of Branches

# Chapter 4

# Case of loops

In this section, we extend our work to a loop represented by a graph $G = (V, E, \delta, \delta_w, \delta_r, \lambda)$ that models its data dependence constraints, such that :

- $V = V_R \cup V_S$ : $V_R$ the set of value nodes (write in a register of the considered type) and the $V_S$ remaining nodes ;
- $E = E_R \cup E_S$ : $E_S$ the set of flow arcs (true dependence through a register of the considered type), and $E_S$ serial arcs (true dependences through memory or other register types) ;
- $\delta$ is the latency function for each arc ;
- $\delta_w$ is the write delay ;
- $\delta_r$ is the read delay ;
- $\lambda$ is the distance of a dependence in term of number of iterations.

The purpose of this section is to calculate an upper-bound of the register saturation for any loop intended for software pipelining (SWP) schedule. Since we do not worry about resource constraints, we assume any SWP that could be built for the precedence constraints modeled by $G$ : any other constraints like resources used to build SWP schedules are a subset of the set of all schedules that could be built for $G$ only.

## 4.1 General Approach

Our idea is to unroll the loop a certain number of times and apply the DAG technique. Then we reroll the DAG to come back to a new graph with some new serialization edges. We can then assure that any SWP schedule of this new graph cannot use more then a certain amount of physical registers.

Unrolling and rerolling are obvious tasks. They are explained below.

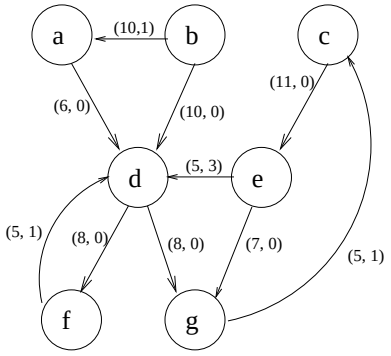
**Definition 4.1 (Unrolling Function)** *Given a graph $G = (V, E, \delta, \delta_w, \delta_r, \lambda)$ modeling precedence constraints in a loop, the unrolling function applied to $G$ with a strictly positive unrolling degree $j$ produces a DAG $D_{\times j}(G) = (V', E', \delta, \delta'_w, \delta'_r)$, called* **unrolled loop**, *such that :*

- $V' = V^0 \cup V^1 \cup \cdots \cup V^{j-1}$ *is the set of nodes in $V$ repeated $j$ times, and*

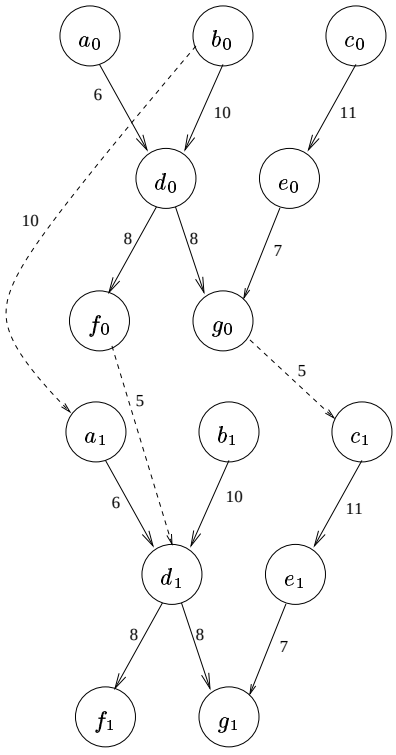
- $V^i = \{u_i / u \in V \wedge \delta'_r(u_i) = \delta_r(u) \wedge \delta'_w(u_i) = \delta_w(u)\} \forall 0 \leq i < j$ *each node in* $V$ *has* $j$ *copies in* $V'$ *;*
- $V'_R = \{u_i / 0 \leq i < j \wedge u \in V_R\}$ *value nodes in* $V'$ *are the copies of value nodes in* $V_R$ *;*
- $V'_S = V' - V'_R$ *;*

- $E' = \{(u_i, v_{i+l}) / u_i \in V^i \wedge v_{i+l} \in V^{i+l} \wedge e = (u, v) \in E \wedge \lambda(e) = l \wedge i + l < j\}$. *We call* $(u_i, v_{i+l})$ *a copy of* $e$. *Each intra-iteration arc in* $E$ *has* $j$ *copies in* $E'$, *and each inter-iteration arc has as many copies as destinations belonging to* $V'$ *;*
- $e' \in E'$ *is a copy of* $e \in E \Longrightarrow \delta'(e') = \delta(e)$.

**Example 4.1.1** *Figure 4.1 gives an example of unrolling. The graph* $G$ *presented in part (a) is unrolled two times producing the DAG in part (b). Each arc* $e$ *of* $G$ *is labeled by the couple* $(\delta, \lambda)$, *and each arc in the DAG* $D_{\times 2}(G)$ *is labeled by* $\delta'$. *We remark that the arc* $(e, d)$ *does not exist in the unrolled loop since the unrolling degree* 2 *is not enough to have the destination* $d_3$ *in the set of nodes.*




*(a) the graph G*




(b) $D_{\times 2}(G)$

Figure 4.1: Unrolling function

The rerolling function is the inverse of unrolling. That is it produces the graph from an unrolled one.

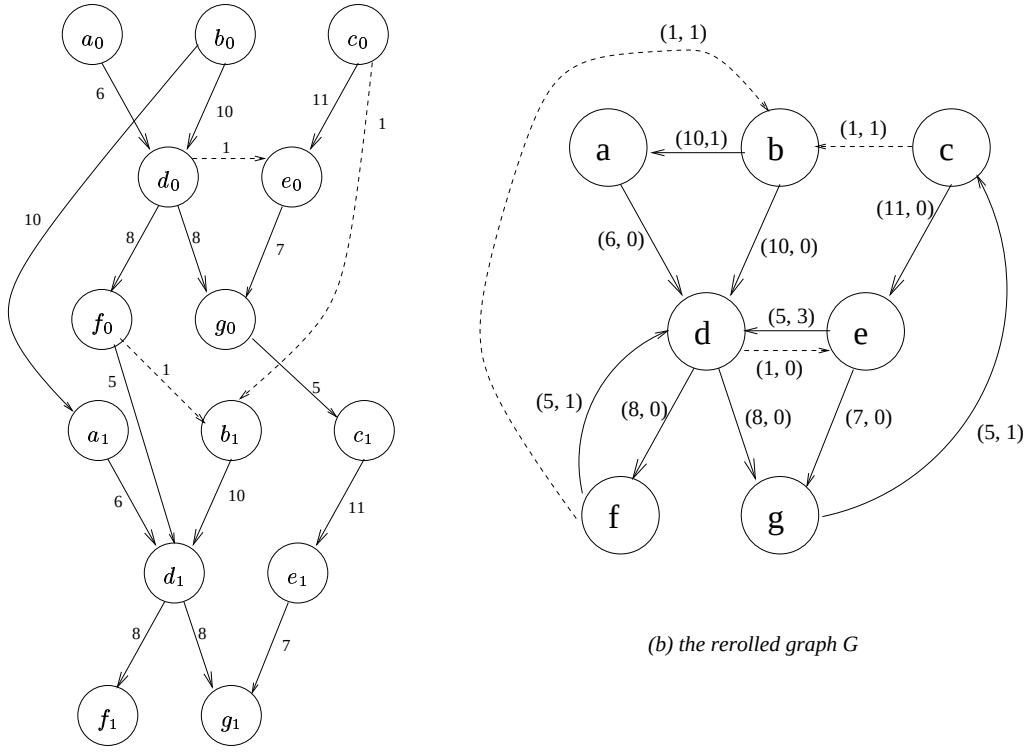
**Definition 4.2 (Rerolling Function)** *Given an unrolled DAG* $D_{\times j}(G) = (V', E', \delta', \delta'_w, \delta'_r)$ *of a graph* $G = (V, E, \delta, \delta_w, \delta_r, \lambda)$ *extended with some new serialization arcs* $E_D \subseteq E'$, *the*

*rerolling function applied to $D_{\times j}(G)$ produces an extended graph $G\backslash^{E_G}$, called* **rerolled graph**, *such that:*

$$E_G = \left\{ e = (u,v)/e' = (u_i, v_{i+l}) \in E_D \wedge \delta(e) = \delta'(e') \wedge \lambda(e) = l \right\}$$

*where $u_i, v_{i+l}$ are the copies of $u, v$.*

**Example 4.1.2** *Figure 4.2 gives an example of rerolling the unrolled loop of the example 4.1.1 after adding the new serialization arcs $(f_0, b_1)$, $(c_0, b_1)$ and $(d_0, e_0)$ (dashed arcs in part a). The rerolled graph is presented in part (b). The new arcs are $(f, b)$, $(c, b)$ and $(d, e)$ with respective latencies and distances $(1, 1)$, $(1, 1)$ and $(1, 0)$.*




(a) $D_{\times 2}(G)$ with new serialization arcs

(b) the rerolled graph G

Figure 4.2: Rerolling function

Now, we only have to define the unrolling degree which is the purpose of the remaining of this chapter. We begin by recalling some notions about software pipelining technique.

## 4.2 Software Pipelining

A software pipelining schedule $\sigma$ of a graph $G = (V, E, \delta, \delta_w, \delta_r, \lambda)$ representing precedence constraints of a loop with $n$ iterations, consists in overlapping the execution of parallel operations belonging to different iterations. The schedule is defined by an **initiation interval** $h$. Each $h$ steps, a new iteration is issued. The schedule is written:

$$\forall u \in V, \forall i \in [0, n[ \qquad \sigma(u, i) = \sigma_u + h \times i$$

where $\sigma(u,i)$ is the schedule of the node $u$ of iteration $i$, and $\sigma_u = \sigma(u,0)$ is the schedule of the node $u$ of the first iteration.

The authors in [WEJS94] model the motif of a SWP schedule as a two dimensional matrix by defining a column number $cn$ and row number $rn$ for each operation. So a SWP schedule is defined by three parameters, we note it $\sigma([rn],[cn],h)$. They define $\sigma$ as:

$$\forall u \in V,\ \forall i \in [0,n[ \qquad \sigma(u,i) = rn(u) + h \times (cn(u) + i)$$

where $cn(u) = \sigma_u\ div\ h$ and $rn(u) = \sigma_u\ mod\ h$.
Graphically (figure 4.3), the row number $rn(u)$ is the step of the execution of $u$. Each instance of an operation $u$ begins its execution $rn(u)$ cycles after the beginning of the motif.

$$\forall u \in V,\ 0 \leq rn(u) \leq h-1$$

The column number $cn(u)$ is the iteration of $u$. Operations having the same column numbers are those scheduled in parallel.

$$\sigma_u = cn(u) \times h + rn(u)$$

We can deduce easily that

$$\forall u \in V \qquad 0 \leq cn(u) \leq \lceil \frac{L_\sigma}{h} \rceil$$

where

$$L_\sigma = \max_{u \in V} \sigma_u \text{ is the schedule time of one iteration}$$

**Definition 4.3 (Valid SWP schedule)** *Given a loop $G = (V,E,\delta,\delta_w,\delta_r,\lambda)$ , a SWP schedule $\sigma([rn],[cn],h)$ is valid iff:*

$$\forall e = (u,v) \in E, \quad rn(v) - rn(u) + \ h(\lambda(e) + cn(v) - cn(u)) \geq \delta(e)$$

It is well known that

$$\sigma([rn_0],[cn_0],h_0) \text{ is optimal } \Longrightarrow h_0 = \max_{\mathcal{C} \text{ a cycle in } G} \left( \frac{\sum_{e \in \mathcal{C}} \delta(e)}{\sum_{e \in \mathcal{C}} \lambda(e)} \right)$$

We note $\lambda(\mathcal{C}) = \sum_{e \in \mathcal{C}} \lambda(e)$ and $\delta(\mathcal{C}) = \sum_{e \in \mathcal{C}} \delta(e)$. We call the cycle $\mathcal{C}_0$ that maximizes this ratio the **critical cycle**.

### 4.2.1 Register Need of a Motif

Since the motif represents a cyclic schedule, some values could be alive for many iterations (see figure 4.4). Then, lifetime intervals become circular around the motif. If $\sigma$ is valid, then

$$\forall e = (u,v) \in V_R,\ \forall i \in [0,n] \qquad \sigma(u,i) + \delta(e) \leq \sigma(v, i + \lambda(e))$$

A value $u_i$ produced at iteration $i$ is killed at date $killm_\sigma(u_i) = \sigma(v, i + \lambda(e)) + \delta_r(v)$ for $e = (u,v) \in E_R$ iff $\forall v' \in Cons(u), e' = (u,v') \in E_R, \forall i \in [0,n]$:

$$rn(v') + h(cn(v') + i + \lambda(e')) + \delta_r(v') \leq rn(v) + h(cn(v) + i + \lambda(e)) + \delta_r(v)$$

cancelling out the $i$, we get the following definition.

**Definition 4.4 (Killing Date in the Motif)** *Le $G = (V, E, \delta, \delta_w, \delta_r, \lambda)$ be a loop and $\sigma([rn], [cn], h)$ a SWP schedule. A value $u \in V_R$ produced in the motif at date $\sigma_u$ is killed at date $killm_\sigma(u) = \sigma_v + h \times \lambda(e) + \delta_r(v)$ for $v \in Cons(u)$ with $e = (u, v) \in E_R$ iff $\forall v' \in Cons(u) / e' = (u, v') \in E_R$ :*

$$rn(v') + h(cn(v') + \lambda(e')) + \delta_r(v') \leq rn(v) + h(cn(v) + \lambda(e)) + \delta_r(v)$$

*Then, the* **lifetime interval** *of $u$, noted $LI_u^\sigma$, is defined by*

$$LI_u^\sigma = ]rn(u) + h \times cn(u) + \delta_w(u),\ rn(v) + h \times (cn(v) + \lambda(e)) + \delta_r(v)]$$

*with $e = (u, v) \in E_R$.*

The **life time** of a value is the amount of time between its definition time ($\sigma_u$) and its killing date :

$$life\_time_\sigma(u) = |LI_u^\sigma| = killm_\sigma(u) - \sigma(u) - \delta_w(u)$$

**Definition 4.5 (Cyclic Life Interval)** *Let $G = (V, E, \delta, \delta_w, \delta_r, \lambda)$ be a loop and $\sigma([rn], [cn], h)$ SWP schedule. The cyclic life interval $CLI_u^\sigma$ of $u \in V_R$ in the motif is the lifetime interval modulo $h$. It is defined by a family $\mathcal{F}_u = \{I_0, \ldots, I_l\}$ of intervals in [0,h[ such that ;*

- *number of intervals*

$$l = \left\lceil \frac{life\_time_\sigma(u)}{h} \right\rceil$$

- *intermediate intervals :* $\forall 0 < j < l, \qquad I_j = [0, h[$

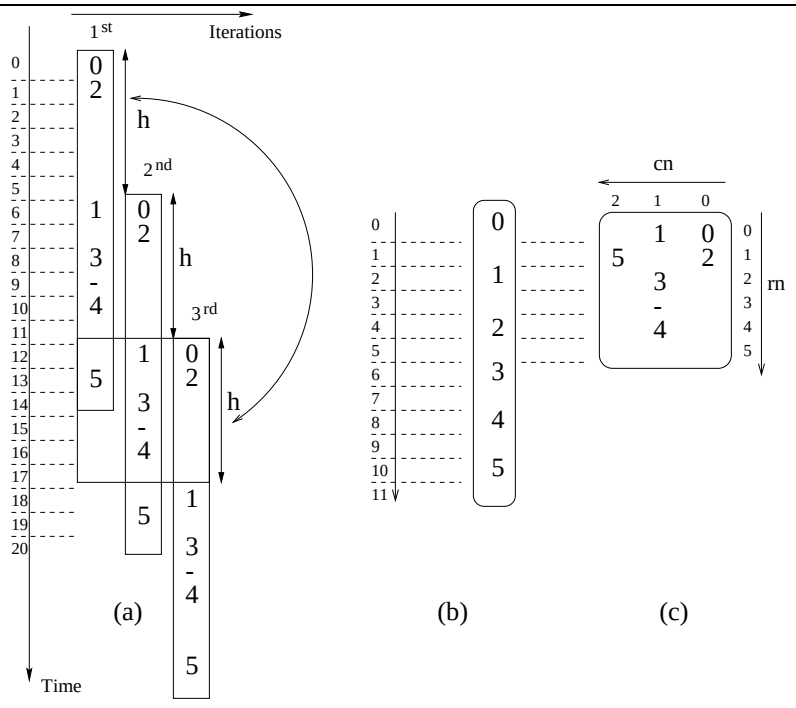



Figure 4.3: (a) Iteration overlaps (b) One iteration schedule (c) Software pipelining motif

- *if* $l > 0$ *then* $I_0 = ]rn(u) + \delta_w(u), h[ \wedge I_l = [0, killm_\sigma(u)]$
- *else* $(l = 0)$ $I_0 = I_l = ]rn(u) + \delta_w(u), killm_\sigma(u)]$

We denote by $\mathcal{LF}$ the set of all circular intervals in a loop:

$$\mathcal{LF} = \bigcup_{u \in V_R} \mathcal{F}_u$$

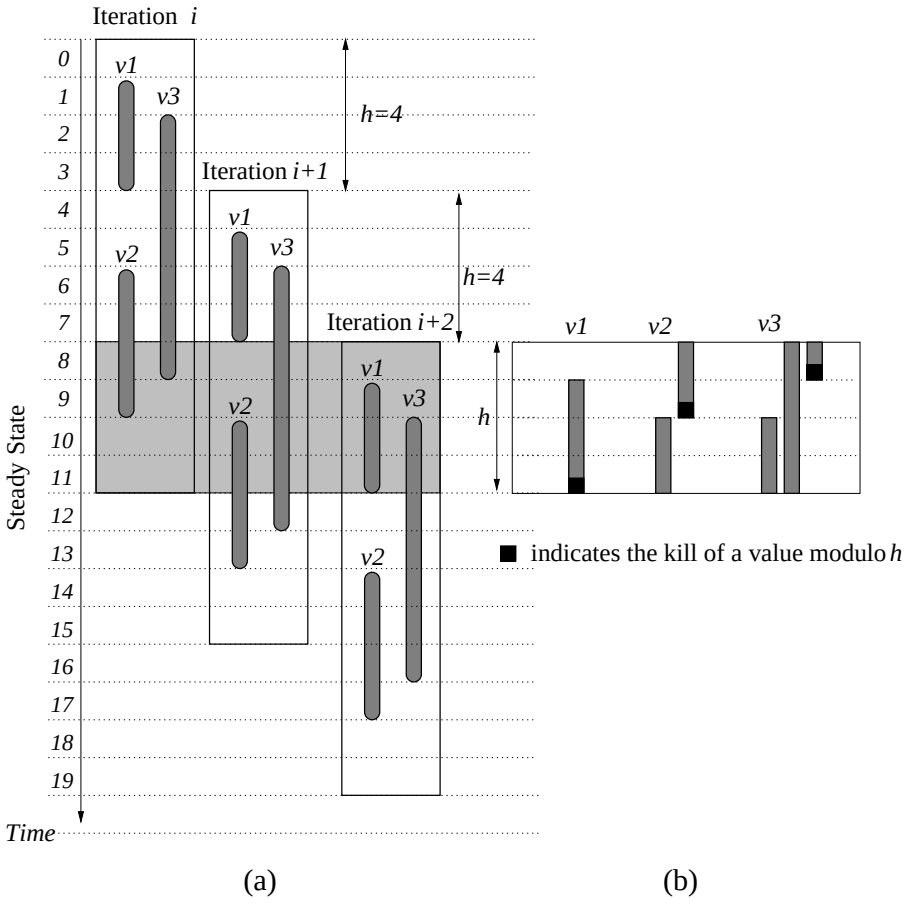



Figure 4.4: (a)Value lifetime intervals in the motif -(b)The family of circular intervals

Then, computing the register need is done by examining the maximum number of overlapped intervals for all the values.

**Definition 4.6 (Register Need of a Motif)** *Given a loop* $G = (V, E, \delta, \delta_w, \delta_r, \lambda)$ *and a SWP schedule* $\sigma([rn], [cn], h)$*, the register need of the SWP motif* $MRN_\sigma(G)$ *is defined by*

$$MRN_\sigma(G) = \max_{0 \leq i < h} |vsa_\sigma(i)|$$

*with*

$$vsa_\sigma(i) = \{I \in \mathcal{LF} / i \in I\} \text{ the overlapped intervals at instant } i$$

Computing $MRN(G)$ has a polynomial complexity $O(h \times |\mathcal{F}|)$, see algorithm 6.

**Algorithm 6** Computing register need in a motif

**Require:** all circular lifetime intervals $\mathcal{LF}$
  $MRN = 0$
  **for** $0 \leq i < h$ **do** {initialization}
    $vsa(i) = 0$
  **end for**
  **for** $0 \leq i < h$ **do** {counting vsa}
    **for all** $I \in \mathcal{LF}$ **do**
      **if** $i \in I$ **then**
        $vsa(i) = vsa(i) + 1$
      **end if**
    **end for**
  **end for**
  **for** $0 \leq i < h$ **do** {searching the maximum vsa}
    **if** $vsa(i) > MRN$ **then**
      MRN=vsa(i)
    **end if**
  **end for**

### 4.2.2 Register Saturation of a Loop

It is the maximum values simultaneously alive that could be generated by a SWP schedule. Formally, we write :

**Definition 4.7** *Given a loop $G = (V, E, \delta, \delta_w, \delta_r, \lambda)$ , the register saturation $MRS(G)$ of this loop is defined by :*

$$\forall \sigma([rn], [cn], h) \text{ a SWP schedule of } G : \quad MRN_\sigma(G) \leq MRS(G)$$

## 4.3 Bounding and Reducing Register Saturation of a loop

### 4.3.1 Loop Unrolling Degree

As introduced in section 4.1, we look for an unrolling degree that permits us exploiting the DAG approach studied in chapter 3. This unrolling degree $j$ must satisfy the condition that the register saturation of the unrolled loop should be greater or equal than the register need produced by any SWP schedule. That is we should unroll the loop enough times to guarantee the following property.

**Definition 4.8 (Valid Unrolling Degree)** *Let $G = (V, E, \delta, \delta_w, \delta_r, \lambda)$ be a loop and a strictly positive unrolling degree $j$. The DAG $D_{\times j}(G) = (V', E', \delta', \delta'_w, \delta'_r)$ is the unrolled loop. $j$ is a valid unrolling degree iff it guarantees the following property :*

$$\forall \text{ valid } \sigma([rn], [cn], h) : \quad MRN_\sigma(G) \leq RS\big(D_{\times j}(G)\big) \tag{4.1}$$

To find a valid unrolling degree that preserves the register saturation property (4.1), we begin by defining the notion of the **first possible value definition** and **last possible value kill** that makes us write lemma 4.1.

**Definition 4.9 (First Possible Value Definition)** *Given a loop $G = (V, E, \delta, \delta_w, \delta_r, \lambda)$ and its body $B = (V, E', \delta, \delta_w, \delta_r)$*[1] *, the first possible value definition of $G$, noted $fd(G)$, defined by:*

$$fd(G) = \min_{u \in V_R} AsSoonAsPossible_B(u) + \delta_w(u)$$

*where*

$$AsSoonAsPoss_B(u) = \max_{v \in Source(B)} LongestPath_B(v, u)$$

$fd(G)$ is the earliest date where a value definition can occur in iteration $i$ of any SWP schedule motif of the loop $G$.

**Definition 4.10 (Last Possible Value Kill)** *Given a loop $G = (V, E, \delta, \delta_w, \delta_r, \lambda)$ and its body $B = (V, E')$, and given a valid initiation interval $h$* [2] *with a schedule time of one iteration $L$, then $\forall \sigma([rn], [cn], h) / \ L_\sigma = L$ we define the last possible value kill, noted $lk^h_L(G)$, as*

$$lk^h_L(G) = \max_{v \in CONS(G)} \left( L - CloseToSink_B(v) + \delta_r(v) + h \times \max_{e=(u,v) \in E_R} \lambda(e) \right)$$

*where*

$$CloseToSink_B(u) = \max_{v \in Sink(B)} LongestPath_B(u, v)$$

*and $CONS(G) = \cup_{u \in V_R} Cons(u)$ is the set of all consumers in $G$.*

In other words, $CloseToSink_B(v)$ defines the minimal amount of execution time between $v$ to the sinks for any schedule. $lk^h_L(G)$ is the latest possible date when a value produced at iteration $i$ in the motif could be consumed in a SWP schedule $\sigma$. The latter must have $h$ as an initiation interval and $L$ as one iteration execution time.

**Lemma 4.1** *Given a loop $G = (V, E, \delta, \delta_w, \delta_r, \lambda)$ then $\forall \sigma([rn], [cn], h)$*

$$\forall u \in V_R \qquad LI^\sigma_u \subseteq ]fd(G), \ lk^h_{L_\sigma}(G)]$$

That means all lifetime intervals of values produced in the motif of $\sigma$ belong to the window $w = ]fd(G), \ lk^h_{L_\sigma}(G)]$. We call it the **observation window**.

[1] $E' = \{e \in E / \ \lambda(e) = 0\}$

[2] $h \geq h_0$

**Proof :**

Given SWP schedule $\sigma([rn],[cn],h)$, the window that contain all value lifetime intervals is bounded by two dates $]d_u, d_v]$ :

1. $d_u = \sigma_u + \delta_w(u)$ which is the earliest definition produced in the motif, that define a value $u$ ;
2. $d_v = \sigma_v + \delta_r(v) + h \times \lambda(e) = killm_\sigma(t)$ which is the latest kill in the motif, such that $t$ is killed by $v$ thanks to the flow arc $e = (t, v)$ ;

We have to prove that $fd(G) \leq d_u \leq d_v \leq lk^h_{L_\sigma}(G)$. Let $B$ be the loop body of $G$. By definition 4.9, $fd(G) \leq d_u$ is obvious :

$$\sigma_u \geq AsSoonAsPossible_B(u) \Longrightarrow \sigma_u + \delta_w(u) \geq AsSoonAsPossible_B(u) + \delta_w(u)$$

We only have to prove that $d_v \leq lk^h_{L_\sigma}(G)$. $v$ is scheduled at the date

$$rn(v) + h \times (cn(v) + \lambda(e)) = \sigma_v + h \times \lambda(e)$$

We know that $\sigma_v \leq L$. But since

$$\forall s \in Sink(B) \quad \sigma_s - \sigma_v \geq LongestPath_B(v, s)$$

Then

$$\sigma_s - \sigma_v \geq \max_{s \in Sink(B)} LongestPath_B(v, s) \Longrightarrow \sigma_v \leq \sigma_s - CloseToSink_B(v)$$

Since $\max_{s \in Sink(B)} \sigma_s = L$, the date $d_v$ is at most equal to the date :

$$d_v \leq L - CloseToSink_B(v) + \delta_r(v) + h \times \lambda(e)$$

And by definition 4.10

$$d_v \leq lk^h_{L_\sigma}(G)$$

⌟

Lemma 4.1 gives a formulation to bound values lifetime intervals produced in the motif. This leads us to the following theorem.

**Theorem 4.1** *Given a loop $G = (V, E, \delta, \delta_w, \delta_r, \lambda)$ and assuming an upper-bound $L_{max}$[3] for iteration execution time that could be produced for any SWP schedule, then the following unrolling degree is valid*

$$j = \max_{u \in CONS(G)} \left( \left\lceil \frac{L_{max} - CloseToSink_B(u) + \delta_r(u)}{h_0} \right\rceil + \max_{e=(v,u) \in E_R} \lambda(e) \right)$$

[3] $\forall \sigma([rn],[cn],h) \qquad L_\sigma \leq L_{max}$

**Proof :**

Given a SWP schedule $\sigma([rn],[cn],h)$ for a loop $G=(V,E,\delta,\delta_w,\delta_r,\lambda)$ , the number of values simultaneously alive in the motif $(]0,h])$ is the number of values simultaneously alive in the observation window (lemma 4.1). Since the ending date of the observation windows is $lk^h_{L_\sigma}(G)$, the number of iterations spanned until this date is (figure 4.5)

$$\left\lceil \frac{lk^h_{L_\sigma}(G)}{h} \right\rceil = \max_{u\in Cons(G)} \left( \left\lceil \frac{L - CloseToSink_B(u) + \delta_r(u)}{h} \right\rceil + \max_{e=(v,u)\in E_R} \lambda(e) \right)$$

That is the register need of $\sigma$ is defined by the cyclic lifetime intervals of the values produced within these iterations. If we generalize to any SWP schedule, the register need could not exceed the amount of values simultaneously alive produced within

$$j = \max_{h,L_\sigma} \left\lceil \frac{lk^h_{L_\sigma}(G)}{h} \right\rceil \text{ iterations}$$

In fact, $j$ defines the further possible iteration when a value could be killed : a value defined at iteration $i$ can be killed at the latest in iteration $i+j$. Then it could not exceed the register saturation of the loop $G$ unrolled $j$ times :

$$j = \max_{u\in Cons(G)} \left( \max_{h,L_\sigma} \left\lceil \frac{L_\sigma - CloseToSink_B(u) + \delta_r(u)}{h} \right\rceil + \max_{e=(v,u)\in E_R} \lambda(e) \right)$$

Since $h \geq h_0$ and assuming an upper bound $L_{max}$ for $L_\sigma$ :

$$j = \max_{u\in Cons(G)} \left( \left\lceil \frac{L_{max} - CloseToSink_B(u) + \delta_r(u)}{h_0} \right\rceil + \max_{e=(v,u)\in E_R} \lambda(e) \right)$$

**Bounding $L_\sigma \leq L_{max}$** Since $L_\sigma$ is defined by resource constraints, in our study we try to give a suitable $L_{max}$ for all SWP schedules. We assume then the worst case where the resource constraints do not permit any parallelism between operations, i.e. they are sequentially scheduled. We choose :

$$L_{max} = \sum_{u\in V} lat(u)$$

as a suitable upper-bound for any $L_\sigma$.

⌟

**Case of Acyclic Loops**

If there is no cycle in the loop $G=(V,E,\delta,\delta_w,\delta_r,\lambda)$ i.e. $h_0 = 0$, then there is no precedence constraint for building a valid cyclic schedule. A SWP schedule tries to put all $V$ operations parallel in the motif (see figure 4.6). A valid lower-bound for initiation interval in this case is $h \geq 1$ and the corresponding valid unrolling degree is :

$$\max_{u\in CONS(G)} \left( L_{max} - CloseToSink_B(u) + \delta_r(u) + \max_{e=(v,u)\in E_R} \lambda(e) \right)$$

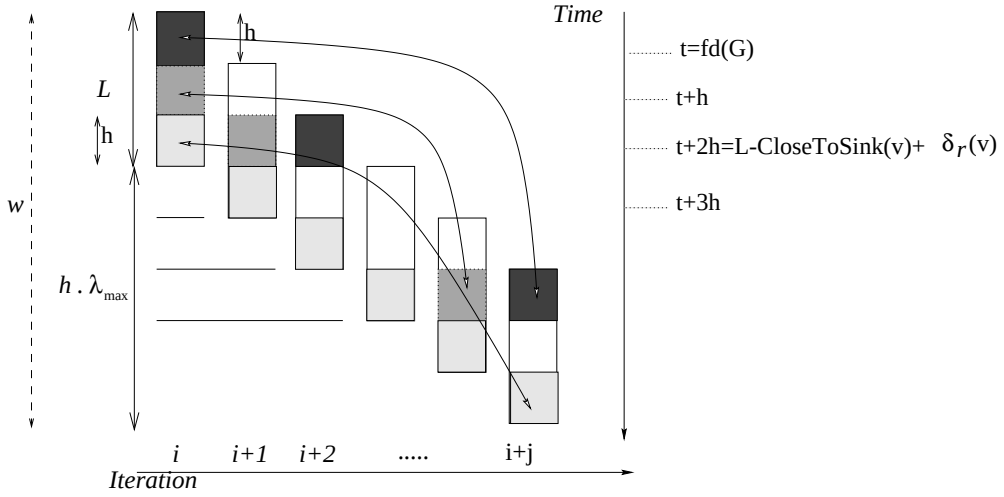
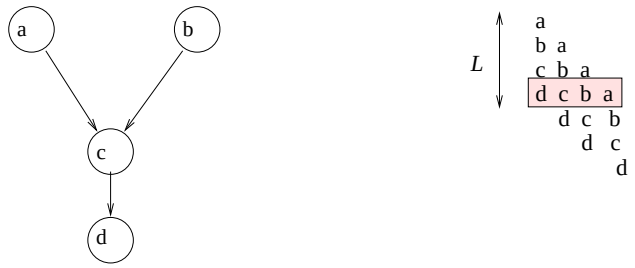



Figure 4.5: Observation window



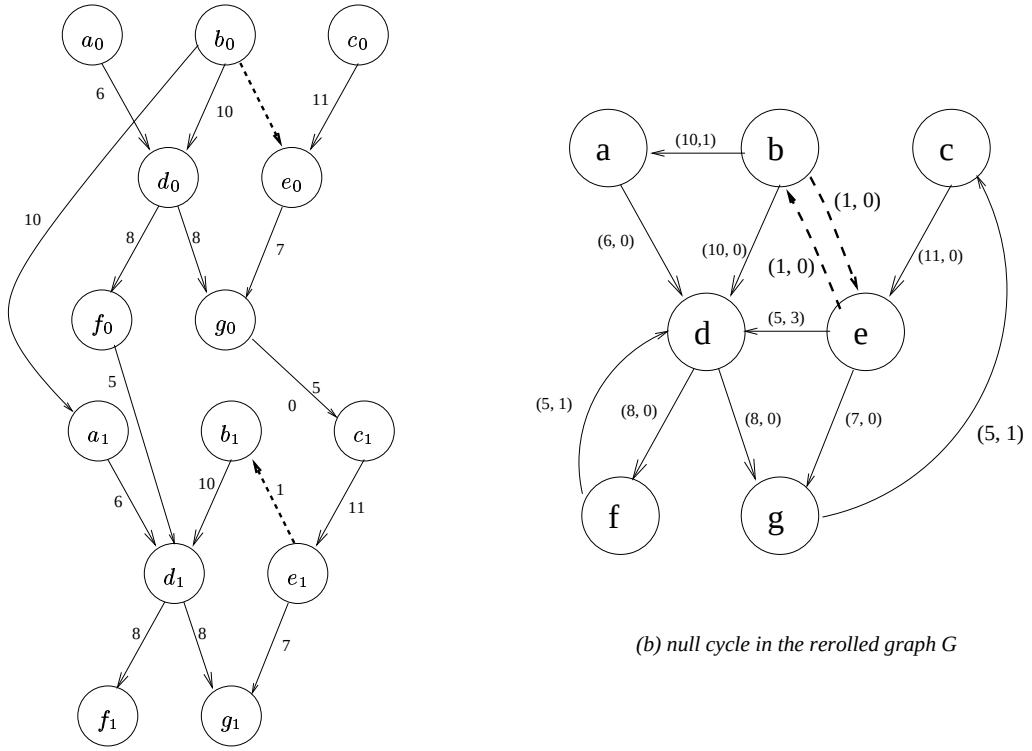

(a) the parallel loop (b) motif of the parallel loop

Figure 4.6: Fully parallel loops

### 4.3.2 Reducing Register Saturation in Unrolled Loops

After computing a valid unrolling degree and after building the unrolled loop, we can use our DAG technique to compute the register saturation. But, some cares must be taken when reducing it because we should keep in our mind that the unrolled DAG will be rerolled to get a cyclic loop. There are two problems that could occur :

1. after rerolling the extended DAG, a null cycle could be introduced in the loop, see figure 4.7 .
2. adding serial arcs between operations that do not permit rerolling. According to Def. 4.2, only serial arcs from $u_i$ to $v_{i+l}$ with $l \geq 0$ could be rerolled, see figure 4.8.




(a) $D_{\times 2}(G)$ with new serialization arcs

Figure 4.7: Problem of Null cycle after rerolling

To avoid the occurrence of these problems, we musr redefine the notion of valid value serialization in the case of unrolled DAGs. Avoiding problem 1 consists in testing if adding a serial arc to the unrolled DAG would introduce cycle in the loop body. Avoiding problem 2 consists in not enabling addition of serial arcs except in the incrementing order, i.e. from node $u_i$ to node $v_{i+l}$ with $l \geq 0$.

**Definition 4.11 (Valid Value Serialization in Unrolled Loops)** *Let be $G = (V, E, \delta, \delta_w, \delta_r, \lambda)$ a loop and $D_{\times j}(G) = (V', E', \delta')$ its unrolled DAG. Let $u_i, v_k \in V'_R$ be two copies of $u, v \in V_R$. An admissible value serialization $u_i \rightarrow v_k$ in the unrolled loop $D_{\times j}$ is valid iff for all introduced serial arcs*[4] *$e = (n_l, v_k)$ :*

[4] arcs from $pkill_{D_{\times j}}(u_i)$ to $v_k$

- *it does not introduce any cycle in $B$ the rerolled loop body:* $\neg(v < n)$ *in* $B$
- *it preserves the lexicographic order of operations:* $l \leq j$

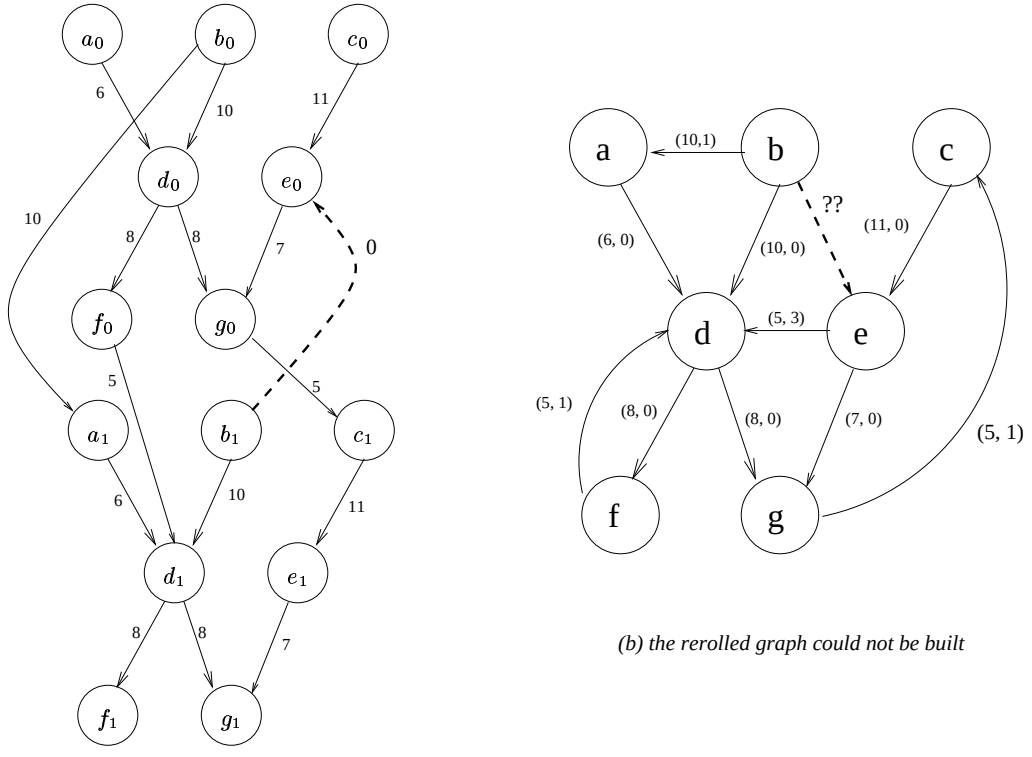
Then, any admissible value serialization introduced in the unrolled loop must satisfy these two conditions to be able to build the new rerolled loop.

## 4.4 Conclusion

Algorithm 7 gives our heuristic that tries to reduce register saturation in case of a loop.

**Algorithm 7** Reducing register saturation in a loop

**Require:** a loop $G = (V, E, \delta, \delta_w, \delta_r, \lambda)$
search the critical cycle $\mathcal{C}_0$ and the minimum initiation interval $h_0$.
compute the valid unrolling degree.
build the unrolled loop $D_{\times j}(G)$
reduce the register saturation in the DAG $D_{\times j}(G)$ {each value serialization must be valid according to Def. 4.11 producing serial arcs $E'$}
reroll the extended graph $D_{\times j}(G)\backslash^{E'}$




(a) $D_{\times 2}(G)$ with a serialization arcs

*(b) the rerolled graph could not be built*

Figure 4.8: Conditions of Def. 4.2 for rerolling not satisfied

# Chapter 5

# Software Implementation and Experimental Validation

In this chapter, we present our experimentation results done on some loops used in [ES96] and [GAG94]. These graphs are presented in appendix F.

In our experimentation, we focus on floating point registers and assume that reading from and writing to registers are done at cycle 0.

## 5.1 Software description

The software is implemented using the LEDA API [MN99]. For displaying graphs interactively, we use xvcg [LS93]. We do not give technical details of our software, we leave this for a further report. We have implemented two object oriented tools:

1. **RS** intended for DAGs: it computes the register saturation and tries to reduce it to an upper bound given by the user. The output is the new extended graph, saturating values, the potential killing DAG $PK(G)$ and the disjoint value DAG $DV_k(G)$;

2. **MRS** intended for loops: it computes an upper bound for the register saturation in the motif and try to reduce it. The output is the new extended loop with its new upper-bound for the register saturation in the motif.

Figure 5.1 gives an example of an original DAG (left side) generated by RS to highlight the original saturating values. These values are in red circles[1], so the register saturation is equal to 5. Other values are green, non values are left gray. Flow arcs are red arrows and serialization arcs are black. In the right side, the extended DAG shows that the register saturation has been reduced to 4.

[1]We apologize for the readers who have a black-and-white version of the paper, since the colors don't come out properly for them; we suggest that they have a look at the postcript file available on the www from the following address: http://www.inria.fr/RRRT/publications-eng.html

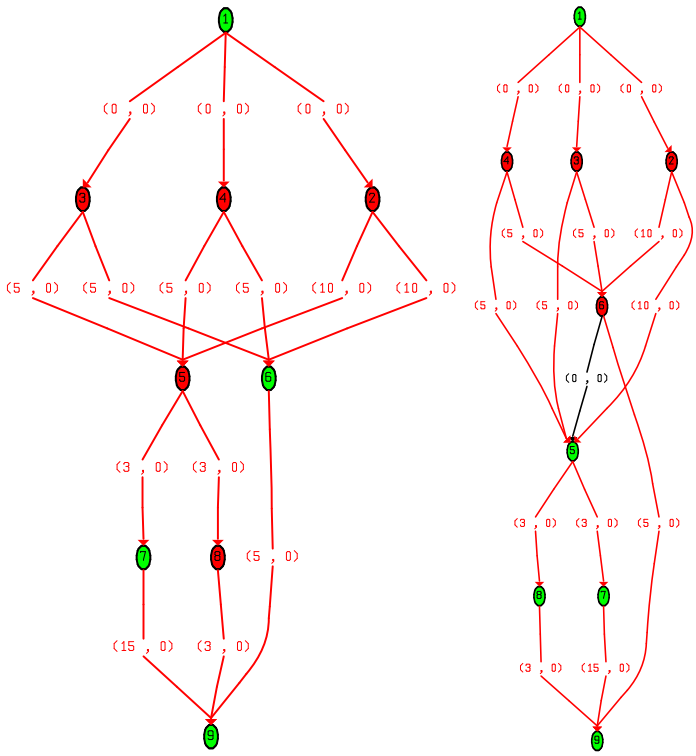



Figure 5.1: Example of reducing the register saturation in a DAG

## 5.2 Experimentation

### 5.2.1 Case of DAGs

The first step of our experimentation is to study the efficiency of our heuristics in the case of DAGs. We take the DAGs of the loop bodies (we ignore inter-iterations arcs) and we try to reduce the register saturation as low as possible[2]. Table 5.1 gives our results. What we see is that the register saturation of the loop bodies are low, ranging from 1 to 8. We have computed easily and manually the register sufficiency (defined in section 3.3 page 35) of each DAG and we find that about 17/26 of these DAGs have $RF = RS$, so we cannot reduce their register saturation. For the remaining DAGs, our heuristic managed in reducing their register saturation.

In a second experience, we unroll loops 10 times to obtain large DAGs. For these DAGs, register saturations range of DAGs from 2 to 80. Only 6 DAGs have a register saturation that exceeds 32 registers. In all cases, our heuristic succeeds in reducing it until exactly 32 registers, and the critical path increased in only one case. Table 5.2 summaries our results. We get linear results if we unroll the loops 20 times and reduce their register saturation until 64 : register saturation range is from 1 to 160, our heuristic succeeds in reducing the register saturation until 64 in the same 6 unrolled loops, and the critical path increases in the same case.

[2]in practice, this is done by giving 1 as a target limit

In a third experience, we unroll the loops with their valid unrolling degrees. Register saturation results in range from 2 to 48. Table 5.4 gives our results. A first remark is that 21/26 of these DAGs have a register saturation lower than 32, so no need to extend them and any schedule cannot produce a spill code. For the five remaining DAGs, their register saturation is lower than 64. Our heuristic manages to reduce their register saturation until exactly 32 registers. But the critical path increases in three of these cases.

Fourth, we use the unrolled loops 20 times and try to reduce their register saturations below 32 registers. In this case, 14 of the 26 unrolled loops exceed 32 saturating values. Our heuristic succeeds in getting the limit 32 in 12 cases, and fails in two cases (we get 39 and 59). The critical path increases in 6 cases. Table 5.3 gives our results.

Finally, we study the amount of ILP lost after adding serial arcs. The amount of ILP in DAGs is the ratio between the number of operations and the critical path :

$$\forall G = (V, E, \delta) \qquad ILP(G) = \left\lceil \frac{|V|}{CriticalPath(G)} \right\rceil$$

The ratio used for expressing the ILP loss is

$$\frac{\text{original ILP} - \text{new ILP}}{\text{original ILP}}$$

We use the limit $\mathcal{R} = 32$ to study the ILP loss. Table 5.5 gives the ILP loss in the case of the unrolled loops 10 times. Table 5.6 is intended for unrolled loops 20 times, and Tab. 5.7 is for the unrolled loops with valid unrolling degree. In all above tables, only cases where register saturation exceeds 32 registers are presented.

### 5.2.2 Case of Loops

The second step of our experimentation is intended for loops. Our purpose is to prevent any software pipelining schedule to exceed the limit of 32 registers. To recall, we proceed by re-rolling unrolled DAGs to get a cyclic graph. Table 5.4 gives us the valid unrolling degree and an upper-bound for the register saturation in the motif. The first remark is that the unrolling degree could be high in the case where the critical cycle is low. Second, any software pipelining schedule for all these loops cannot use more than 64 registers. Third, 21 of these 26 loops have a register saturation that does not exceed 32. Since the unrolling degree is valid, we are sure that any software pipelining schedule does not exceed 32 values simultaneously alive. For the five remaining loops, table 5.8 gives the new upper-bound of the register saturation in the motif after rerolling the extended DAGs. We remark that the upper-limit of MRS of the rerolled loop reduces compared to the register saturation of the unrolled DAG. The reason is that the rerolling function makes cyclic some serial arcs : a single serial arc in the unrolled DAG become a cyclic arc in the loop. In all cases, our heuristic succeeds in reducing it below the limit 32. The drawback is that in three cases, the critical cycle increases. And in another case (spec-spice-loop7), our heuristic introduces a cycle in the loop while the original graph is acyclic : the reason is that the rerolling function could produce cyclic serializations which are difficult to avoid in reduced complexity.

| | Before reduction | | After reduction | |
|---|---|---|---|---|
| **loop body** | **RS** | **Critical Path** | **min RS obtained** | **Critical Path** |
| Lin-ddot | 2 | 4 | 2 | 4 |
| liv-loop1 | 3 | 8 | 2 | 10 |
| liv-loop23 | 8 | 12 | 4 | 16 |
| liv-loop5 | 2 | 5 | 2 | 5 |
| spec-dod-loop1 | 5 | 22 | 4 | 25 |
| spec-dod-loop2 | 4 | 23 | 3 | 23 |
| spec-dod-loop3 | 4 | 24 | 3 | 24 |
| spec-dod-loop7 | 1 | 35 | 1 | 35 |
| spec-fppp-loop1 | 2 | 20 | 1 | 20 |
| spec-spice-loop10 | 2 | 3 | 2 | 3 |
| spec-spice-loop1 | 1 | 2 | 1 | 2 |
| spec-spice-loop2 | 3 | 8 | 3 | 8 |
| spec-spice-loop3 | 2 | 5 | 2 | 5 |
| spec-spice-loop4 | 6 | 10 | 6 | 10 |
| spec-spice-loop5 | 1 | 2 | 1 | 2 |
| spec-spice-loop6 | 3 | 21 | 2 | 21 |
| spec-spice-loop7 | 3 | 19 | 2 | 19 |
| spec-spice-loop8 | 2 | 3 | 2 | 3 |
| spec-tom-loop1 | 6 | 24 | 4 | 24 |
| whet-cycle4_1 | 1 | 3 | 1 | 3 |
| whet-cycle4_2 | 1 | 3 | 1 | 3 |
| whet-cycle4_4 | 1 | 3 | 1 | 3 |
| whet-cycle4_8 | 1 | 3 | 1 | 3 |
| whet-loop1 | 3 | 16 | 3 | 16 |
| whet-loop2 | 2 | 24 | 2 | 24 |
| whet-loop3 | 4 | 4 | 4 | 4 |

Table 5.1: DAGs of the loop bodies ($\mathcal{R} = 1$)

| | Before reduction | | After reduction | |
|---|---|---|---|---|
| **loop** | **RS** | **Critical Path** | **RS reduced** | **Critical Path** |
| Lin-ddot | 20 | 13 | 20 | 13 |
| liv-loop1 | 21 | 44 | 21 | 44 |
| liv-loop23 | 80 | 84 | 32 | 100 |
| liv-loop5 | 20 | 32 | 20 | 32 |
| spec-dod-loop1 | 41 | 220 | 32 | 220 |
| spec-dod-loop2 | 40 | 212 | 32 | 212 |
| spec-dod-loop3 | 40 | 204 | 32 | 204 |
| spec-dod-loop7 | 10 | 44 | 10 | 44 |
| spec-fppp-loop1 | 12 | 200 | 12 | 220 |
| spec-spice-loop10 | 11 | 30 | 11 | 30 |
| spec-spice-loop1 | 10 | 11 | 10 | 11 |
| spec-spice-loop2 | 30 | 17 | 30 | 17 |
| spec-spice-loop3 | 2 | 59 | 2 | 59 |
| spec-spice-loop4 | 44 | 100 | 32 | 100 |
| spec-spice-loop5 | 10 | 29 | 10 | 29 |
| spec-spice-loop6 | 30 | 39 | 30 | 39 |
| spec-spice-loop7 | 30 | 20 | 30 | 20 |
| spec-spice-loop8 | 20 | 3 | 20 | 3 |
| spec-tom-loop1 | 43 | 222 | 32 | 222 |
| whet-cycle4_1 | 1 | 39 | 1 | 39 |
| whet-cycle4_2 | 2 | 19 | 2 | 19 |
| whet-cycle4_4 | 4 | 11 | 4 | 11 |
| whet-cycle4_8 | 8 | 7 | 8 | 7 |
| whet-loop1 | 6 | 169 | 6 | 169 |
| whet-loop2 | 20 | 78 | 20 | 78 |
| whet-loop3 | 4 | 49 | 4 | 49 |

Table 5.2: DAGs of unrolled loops 10 times ($\mathcal{R} = 32$)

| | Before reduction | | After reduction | |
|---|---|---|---|---|
| **loop** | **RS** | **Critical Path** | **RS reduced** | **Critical Path** |
| Lin-ddot | 40 | 23 | 32 | 23 |
| liv-loop1 | 41 | 84 | 32 | 95 |
| liv-loop23 | 160 | 164 | 59 | 218 |
| liv-loop5 | 40 | 62 | 32 | 62 |
| spec-dod-loop1 | 81 | 440 | 39 | 456 |
| spec-dod-loop2 | 80 | 422 | 32 | 422 |
| spec-dod-loop3 | 40 | 204 | 32 | 204 |
| spec-dod-loop7 | 20 | 54 | 20 | 54 |
| spec-fppp-loop1 | 22 | 400 | 22 | 400 |
| spec-spice-loop10 | 21 | 60 | 21 | 60 |
| spec-spice-loop1 | 20 | 21 | 20 | 21 |
| spec-spice-loop2 | 60 | 27 | 32 | 35 |
| spec-spice-loop3 | 2 | 119 | 2 | 119 |
| spec-spice-loop4 | 84 | 200 | 32 | 200 |
| spec-spice-loop5 | 20 | 59 | 20 | 59 |
| spec-spice-loop6 | 60 | 59 | 32 | 59 |
| spec-spice-loop7 | 60 | 20 | 32 | 47 |
| spec-spice-loop8 | 40 | 3 | 32 | 17 |
| spec-tom-loop1 | 83 | 422 | 32 | 422 |
| whet-cycle4_1 | 1 | 79 | 1 | 79 |
| whet-cycle4_2 | 2 | 39 | 2 | 39 |
| whet-cycle4_4 | 4 | 19 | 4 | 19 |
| whet-cycle4_8 | 8 | 11 | 8 | 11 |
| whet-loop1 | 6 | 339 | 6 | 339 |
| whet-loop2 | 40 | 138 | 40 | 138 |
| whet-loop3 | 4 | 99 | 4 | 99 |

Table 5.3: DAGs of unrolled loops 20 times ($\mathcal{R} = 32$)

| | Before reduction | | | After reduction | |
|---|---|---|---|---|---|
| **Unrolled loop** | **Unrolling Degree** | **RS** | **Critical Path** | **RS** | **Critical Path** |
| Lin-ddot | 7 | 14 | 10 | 14 | 10 |
| liv-loop1 | 4 | 9 | 20 | 9 | 20 |
| liv-loop23 | 6 | 48 | 52 | 32 | 52 |
| liv-loop5 | 3 | 6 | 11 | 6 | 11 |
| spec-dod-loop1 | 3 | 13 | 66 | 13 | 66 |
| spec-dod-loop2 | 2 | 8 | 44 | 8 | 44 |
| spec-dod-loop3 | 2 | 8 | 44 | 8 | 44 |
| spec-dod-loop7 | 35 | 35 | 69 | 32 | 88 |
| spec-fppp-loop1 | 2 | 4 | 40 | 4 | 40 |
| spec-spice-loop10 | 2 | 3 | 6 | 3 | 6 |
| spec-spice-loop1 | 3 | 3 | 4 | 3 | 4 |
| spec-spice-loop2 | 13 | 39 | 20 | 32 | 20 |
| spec-spice-loop3 | 2 | 2 | 11 | 2 | 11 |
| spec-spice-loop4 | 6 | 28 | 60 | 28 | 60 |
| spec-spice-loop5 | 1 | 1 | 2 | 1 | 2 |
| spec-spice-loop6 | 14 | 42 | 47 | 32 | 47 |
| spec-spice-loop7 | 21 | 63 | 20 | 32 | 50 |
| spec-spice-loop8 | 6 | 12 | 3 | 12 | 3 |
| spec-tom-loop1 | 3 | 15 | 68 | 15 | 68 |
| whet-cycle4_1 | 1 | 1 | 3 | 1 | 3 |
| whet-cycle4_2 | 2 | 2 | 3 | 2 | 3 |
| whet-cycle4_4 | 4 | 4 | 3 | 4 | 3 |
| whet-cycle4_8 | 8 | 8 | 3 | 8 | 3 |
| whet-loop1 | 3 | 6 | 50 | 6 | 50 |
| whet-loop2 | 5 | 10 | 48 | 10 | 48 |
| whet-loop3 | 3 | 4 | 14 | 4 | 14 |

Table 5.4: DAGs of the unrolled loops with valid unrolling degrees ($\mathcal{R} = 32$)

| **Unrolled loop** | **original ILP** | **new ILP** | **ILP loss** |
|---|---|---|---|
| liv-loop23 | 3 | 2 | 33.33% |
| spec-dod-loop1 | 1 | 1 | 0% |
| spec-dod-loop2 | 1 | 1 | 0% |
| spec-dod-loop3 | 1 | 1 | 0% |
| spec-spice-loop4 | 2 | 2 | 0% |
| spec-tom-loop1 | 1 | 1 | 0% |

Table 5.5: ILP loss in the unrolled loops 10 times ($\mathcal{R} = 32$)

| Unrolled loop | original ILP | new ILP | ILP loss |
|---|---|---|---|
| Lin-ddot | 4 | 4 | 0% |
| liv-loop1 | 3 | 2 | 33.33% |
| liv-loop23 | 3 | 2 | 33.33% |
| liv-loop5 | 2 | 2 | 0% |
| spec-dod-loop1 | 1 | 1 | 0% |
| spec-dod-loop2 | 1 | 1 | 0% |
| spec-dod-loop3 | 1 | 1 | 0% |
| spec-spice-loop2 | 7 | 6 | 14.28% |
| spec-spice-loop4 | 2 | 2 | 0% |
| spec-spice-loop6 | 3 | 3 | 0% |
| spec-spice-loop7 | 5 | 3 | 40% |
| spec-spice-loop8 | 27 | 7 | 74% |
| spec-tom-loop1 | 2 | 2 | 0% |

Table 5.6: ILP loss in the unrolled loops 20 times ($\mathcal{R} = 32$)

| Unrolled loop | original ILP | new ILP | ILP loss |
|---|---|---|---|
| liv-loop23 | 3 | 3 | 0% |
| spec-dod-loop7 | 3 | 2 | 33.33% |
| spec-spice-loop2 | 6 | 6 | 0% |
| spec-spice-loop6 | 2 | 2 | 0% |
| spec-spice-loop7 | 6 | 3 | 50% |

Table 5.7: ILP loss in the unrolled loops with valid unrolling degree ($\mathcal{R} = 32$)

The last experience was intended for studying the ILP loss. In the loop case, we define the ILP as the ratio between the number of operations in the loop body and $h_0$:

$$\forall G = (V, E, \delta, \lambda) \qquad ILP(G) = \left\lceil \frac{|V|}{\lceil CriticalCycle(G) \rceil} \right\rceil$$

Table 5.9 shows that the ILP loss for some extended loops is high. The reason is that since rerolling function makes cyclic each serial arc in the unrolled DAG, this produces obsolete serializations. The second reason is that it could introduce new cycles in the loop, and then it could create a new critical cycle.

| | Before reduction | | After reduction | |
|---|---|---|---|---|
| **Loop** | **MRS$\leq$** | **Critical Cycle** | **MRS $\leq$** | **Critical Cycle** |
| Lin-ddot | 14 | 1/1 | 14 | 1/1 |
| liv-loop1 | 9 | 4/1 | 9 | 4/1 |
| liv-loop23 | 48 | 8/1 | 12 | 16/1 |
| liv-loop5 | 6 | 3/1 | 6 | 3/1 |
| spec-dod-loop1 | 13 | 22/1 | 13 | 22/1 |
| spec-dod-loop2 | 8 | 21/1 | 8 | 21/1 |
| spec-dod-loop3 | 8 | 20/1 | 8 | 20/1 |
| spec-dod-loop7 | 35 | 1/1 | 3 | 18/1 |
| spec-fppp-loop1 | 4 | 20/1 | 4 | 20/1 |
| spec-spice-loop10 | 3 | 3/1 | 3 | 3/1 |
| spec-spice-loop1 | 3 | 1/1 | 3 | 1/1 |
| spec-spice-loop2 | 39 | 1/1 | 27 | 2/1 |
| spec-spice-loop3 | 2 | 6/1 | 2 | 6/1 |
| spec-spice-loop4 | 28 | 10/1 | 28 | 10/1 |
| spec-spice-loop5 | 1 | 3/1 | 1 | 3/1 |
| spec-spice-loop6 | 42 | 2/1 | 29 | 2/1 |
| spec-spice-loop7 | 60 | 0/1 | 15 | 17/3 |
| spec-spice-loop8 | 12 | 0/1 | 12 | 0/1 |
| spec-tom-loop1 | 15 | 22/1 | 15 | 22/1 |
| whet-cycle4_1 | 1 | 4/1 | 1 | 4/1 |
| whet-cycle4_2 | 2 | 4/2 | 2 | 4/2 |
| whet-cycle4_4 | 4 | 4/4 | 4 | 4/4 |
| whet-cycle4_8 | 8 | 4/8 | 8 | 4/8 |
| whet-loop1 | 6 | 17/1 | 6 | 17/1 |
| whet-loop2 | 10 | 6/1 | 10 | 6/1 |
| whet-loop3 | 4 | 5/1 | 4 | 5/1 |

Table 5.8: Reducing MRS in loops below 32 registers (limit $\mathcal{R} = 32$)

| Loop | original ILP | new ILP | ILP loss |
|---|---|---|---|
| liv-loop23 | 3 | 2 | 33.33%% |
| spec-dod-loop7 | 4 | 1 | 75% |
| spec-spice-loop2 | 9 | 5 | 44.44% |
| spec-spice-loop6 | 3 | 2 | 33.33% |
| spec-spice-loop7 | 5 | 1 | 80% |

Table 5.9: ILP loss in the extended loops

# Chapter 6

# Related Work

## 6.1 Scheduling under Register Constraints

A lot of work has been done in combining code scheduling with register allocation in DAGs. The authors in [GH88] give two solutions: the first consists in code scheduling under register constraints for pipelined processors, so the proposed heuristic is driven by code scheduling. The second solution is driven by register allocation, where the DDG is modified whenever a register must be reused: by considering all registers as candidate, one suitable register is chosen. In [BEH91] three solutions are proposed. The first one consists in putting a limit on values simultaneously alive within basic blocs; the second is driven by the register allocation where it estimates a scheduling cost to assign registers to values; the last uses a pre-analysis to estimate the limit used in the first solution. In [Pin93], the author needs a first schedule to build a parallelizable interference graph. She proves that an optimal coloring (with a number of colors do not exceed the amount of registers) of this graph leads to an optimal register allocation without introducing false dependencies. In [Bra94], the author investigates the relationship between graph-coloring register allocation and and ILP scheduling. He gives a new framework consisting in using information from an initial register assignment to remove false dependencies. It tries to create a compromise between advantages of first and last register allocation schemes.

In case of loops, there is a lot of heuristics that combine SWP scheduling under register constraints. An optimal solution using integer programming is studied in [ES96]. In [Huf93], the authors propose a heuristic for SWP that tries to minimize value lifetimes, hoping that this would limit the register need. In [WKE95] a method for building SWP schedules uses a register requirement graph to estimate dynamically the register need. This gives a cost for operation scheduling. The method proposed in [SC96] tries to modify the SWP motif without increasing its length ($h$). Their aim is to reduce the maximal amount of values simultaneously alive . The heuristics studied in [LVA95, LGAV96, Llo96] try to reduce the SWP motif length and the value lifetimes. An operation is scheduled as late as possible or as soon as possible in the motif according to the schedule of their predecessors or successors.

All the above techniques try to build an optimal schedule without exceeding a limit of values simultaneously alive in order to keep these values in physical registers. In our work we do not construct a schedule, we only use precedence constraints. So, any schedule heuristic could be used.

The dual notion of register saturation, called register sufficiency, is studied in [AKR91]. Given a DAG, the authors give a heuristic that find the minimum register needed to complete the computation is $O(log^2|V|)$ factor of the optimal. Register sufficiency tries to build family of schedules with minimal registers, although some DAGs do not need more than the amount physical register. This leads in increasing the critical path and decreasing the amount of ILP.

## 6.2 Similar Work : URSA

### 6.2.1 Maximizing Register Need with URSA

The minimum killing set technique [BGS93] tries to saturate the register need by keeping values alive as late as possible : the authors proceed by keeping as many children alive as possible by computing the minimum set that kills all the parent's values. Formally, the minimum killing set is defined by :

**Definition 6.1 (Minimum Killing Set (MKS))** *:*
*Given a DAG $G = (V, E, \delta, \delta_w, \delta_r)$, the minimum killing set of a connected bipartite component $cb = (S, T, E_{cb})$ in the potential killing DAG $PK(G)$ is a subset $T' \subseteq T$, such that*

1. *covering constraints*

$$\bigcup_{t \in T'} \Gamma^-_{PK(G)}(t) = S$$

2. *minimizing constraints*

$$\min |T'|$$

Finding an MKS is NP-complete. In this section, we prove that the minimum killing set problem does not saturate the register need, even if we have an optimal solution. The problem is that the minimum killing set saturate the register need only in a local bipartite component of the potential killing DAG, and not for the whole DDG.

First, even if we have an optimal solution, the minimum killing set does not give any indication about choosing a killing function. Figure 3.6 in page 17 shows the case where the optimal solution is $\{t_1, t_2, t_3\}$. If we are not careful, we can produce the non valid killing function presented in that figure.

Figure 6.1 describes the first drawback when there is more than one solution to the minimum killing set. Part $(a)$ describes the DDG (for simplicity, here all nodes are value nodes i.e. $\in V_R$ and all edges are flow i.e. $\in E_R$). When computing the minimum killing set of $\{a, b, c\}$, we have the choice between two solutions $\{d\}$ or $\{e\}$. If we choose the former, the disjoint value DAG in part $(b)$ shows that there are 4 registers (the maximal antichain is $\{a, b, c, e\}$, with $k(d) = f$). But if we choose the latter, the disjoint value DAG presented in part $(c)$ show us that there is 5 registers (the maximal antichain is $\{a, b, c, d, g\}$).

The second example in figure 6.2 shows that even if we have one solution for the minimum killing set problem, it does not saturate the register need. For the DAG described part $(a)$, the

minimum killing set of $\{a, b, c\}$ is $\{d\}$. The disjoint value DAG in part $(b)$ shows us that there are 5 values simultaneously alive (the maximal antichain is $\{a, b, c, h, e\}$, with $k(d) = f$). But if we choose the non minimal solution $\{e, h\}$, the disjoint value DAG in part $(c)$ shows that there are 6 values simultaneously alive (the maximal antichain is $\{a, b, c, d, g, i\}$).

The third drawback arises because of their DAG model. The authors do not make difference between nodes, arcs and register types: all their nodes are values and all their arcs are flow, then only one possible register type can be used. Figure 6.3 shows an example. The bold circles refer to value nodes. If we choose the minimum killing set $\{e\}$, the disjoint value DAG in part $(b)$ shows 3 values simultaneously alive . But if we choose another killing set $\{d, f\}$ which is not minimal, the disjoint value DAG in part $(c)$ shows 4 values simultaneously alive . So the minimum killing set does not guarantee the register saturation.

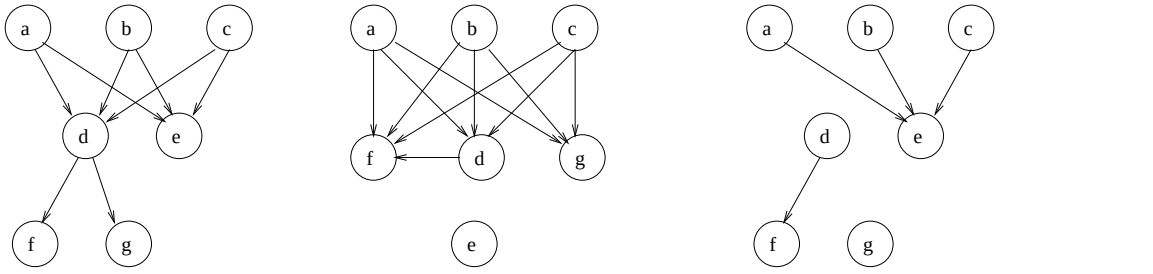



Figure 6.1: The first drawback of minimum killing set technique

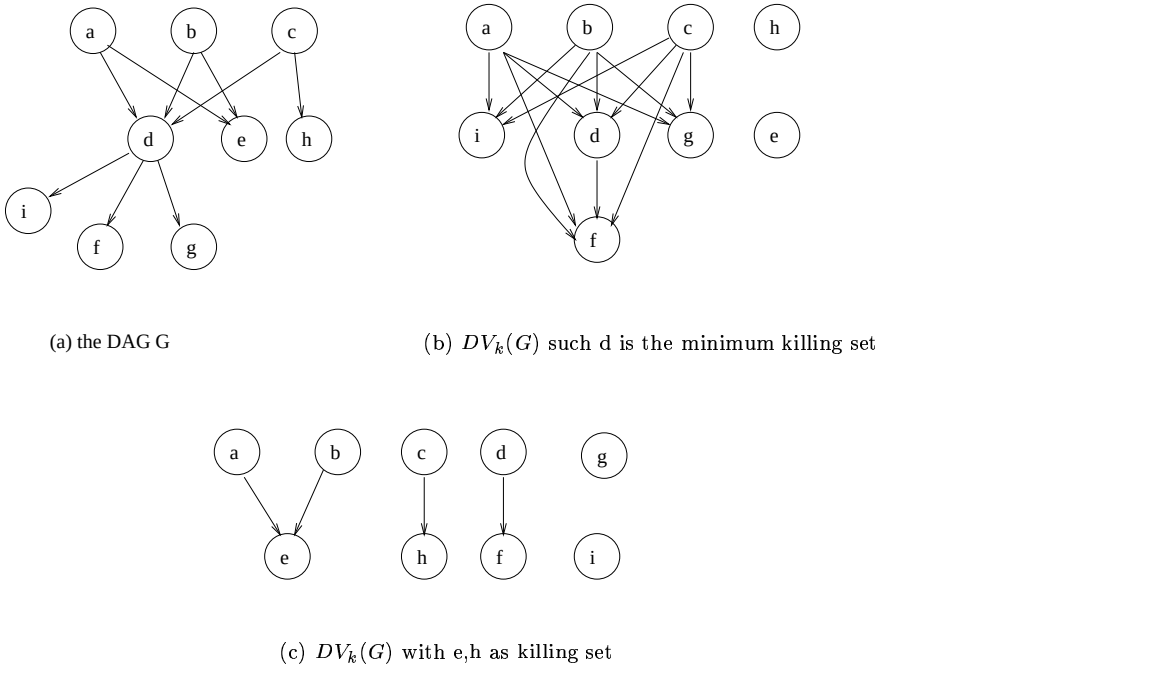



Figure 6.2: The second drawback of minimum killing set technique

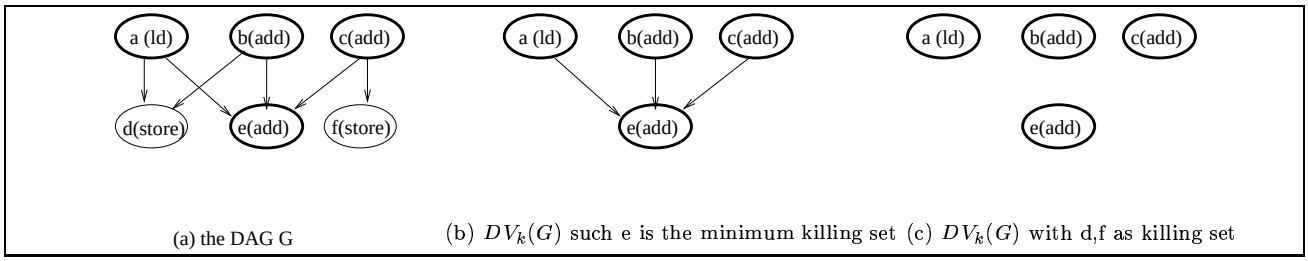



Figure 6.3: The third drawback of minimum killing set technique

### 6.2.2 Reducing Register Use with URSA

The authors give a heuristic in [BGS92] to reduce the register saturation. They use serializations like in our approach, but rather than serializing two values they serialize two sub-DAGs. They look for two sub-DAGs such that the *local* register saturation of the second does not exceed the limit $\mathcal{R}$. And after, they serialize it after the first one. They do not provide an algorithm to find two suitable sub-DAGs. However, their approach should be more complex than our heuristic because searching for a suitable sub-DAG is more complex than searching for a suitable single value node.

# Chapter 7

# Conclusion

In this report, we present our approach to avoid or reduce spill code that could be generated for any schedule and for any resource constraints. We proceed first by computing the saturating values (the maximum amount of values potentially alive). If the amount of these values exceeds the number of physical registers, we add some serial arcs into the the graph to reduce the register saturation. Since the problem of computing saturating values is NP-complete, we give a polynomial-time heuristic to solve this problem. Nevertheless, we get an optimal solution for certain graphs: those that have a potential killing tree. We also give a heuristic that chooses the best suitable value serialization to reduce the register saturation and keep it under a limit (number of physical registers) without increasing the critical path if possible. It fails if the register sufficiency is greater that this limit (since the amount values simultaneously alive is always greater than the register sufficiency for any schedule). However, it produces an extended graph with reduced register saturation, so the spill code generated by a schedule is consequently reduced.

Branches are taken into account by building a DAG for each possible path in the CFG. This is done to get exact flow dependencies in case where more than one operation can define a value. Since static speculation technique can introduce new recovering operations, the register saturation analysis in case of branch must be done after speculation.

We extend our work to loops. Our purpose is to avoid and reduce register saturation to control the amount of values simultaneously alive produced by any SWP schedule. We give a heuristic consisting in unrolling, applying the DAG technique and then rerolling. The valid unrolling degree could be large, but since we reroll the graph, there is no more code growth anymore. If the heuristic succeeds in keeping the register saturation under the limit (number of physical registers), then any SWP schedule cannot generate spill code. If not, the spill code is reduced. Since the rerolling function could introduce new cycles in the loop, new critical cycle could be introduced. We think that our loop approach for reducing register saturation can be improved by avoiding to unroll and reroll the loop.

Our approach is valid for both scheduling semantics (UAL and NUAL). For the latter, we define graph transformations that add intermediate nodes and arcs between operations in order to produce valid schedules in NUAL semantics (guarantee a minimum latency). Only few notions have been redefined and adapted to apply our approach in the NUAL case.

Our perspective is intended for loops. We will study the manner of computing and reducing the register saturation in the motif (MRS) rather than bounding it by the register saturation of the unrolled DAG. The first reason is that the unrolling degree could be high, producing huge DAGs. The second reason is that the rerolled function could introduce new cycles in the loop and some obsolete serial arcs[1] reducing then the ILP. We think that working directly on the cyclic graph is better than working on the unrolled one.

[1]since a serial arc in the unrolled DAG become cyclic in the rerolled loop

# Appendix A

# The Set of Valid Schedules

**Algorithm 8** $\mathcal{LL}(G, \sigma)$

```
if G = (φ, φ) then
  print σ
else
  n=|Source(G)|
  for p=1,n do
    for all S ∈ C^p(Source(G)) do {all combination of p sources}
      call LL(G − S, σ · S)
    end for
  end for
end if
```

Algorithm 8 gives a recursive procedure $\mathcal{LL}(G, \sigma)$ that computes $\Sigma(G)$ by using the first call $\mathcal{LL}(G, \phi)$. Unfortunately, its complexity is not supportable. This algorithm traverses the whole DAG[1] for building one valid schedule, so the complexity is:

$$O\Big((|V| + |E|) \times |\mathcal{LL}(G)|\Big) \leq O\Big((|V| + |E|) \times |V|^{|V|}\Big)$$

We can deduce easily algorithm 9 that computes the set of all valid restricted schedules.

[1] the complexity of visiting all the DAG is $O(|V| + |E|)$

**Algorithm 9** $\overline{\mathcal{LL}}^r(G, \sigma)$

**if** $G = (\phi, \phi)$ **then**
  print $\sigma$
**else**
  n=$|Source(G)|$
  **for** p=1, **min**$(n, r)$ **do**
    **for all** $S \in C^p\big(Source(G)\big)$ **do** {all combination of $p$ sources}
      call $\overline{\mathcal{LL}}^r(G - S, \sigma \cdot S)$
    **end for**
  **end for**
**end if**

# Appendix B

# Constructing Potential Killing DAGs

**Algorithm 10** building $PK\big(G=(V,E)\big)$

$G'.V = G.V$ {we put all the nodes of G in the potential killing DAG}
**for all** arcs $(u,v) \in G.E_R$ **do** {we construct the consumer set of each value node}
  $Cons(u) = Cons(u) \cup \{v\}$
**end for**
$G_c = transitive\_closure(G)$ { permit to test precedence relation between consumers}
**for all** value nodes $u \in G.V_R$ **do** {we compute the potential killing operations}
  **for all** nodes $v \in Cons(u)$ **do**
    **if** $\Gamma^+_{G_c}(v) \cap Cons(u) = \phi$ **then**
      $pkill(u) = pkill(u) \cup \{v\}$
    **end if**
  **end for**
**end for**
**for all** value nodes $u \in G'.V_R$ **do** {we add the potential kill relation in $G'$}
  **for all** nodes $v \in pkill(u)$ **do**
    $G'.E = G'.E \cup \{(u,v)\}$
  **end for**
**end for**
return $G'$

The complexity of algorithm 10 is dominated by the complexity of the transitive closure algorithm. We assume that the reader is familiar with such algorithms [CLR90]. There are two well-known algorithms:

- Roy-Warshall algorithm in $O(|V|^3)$;
- Goralcicova-Koubek algorithm in $O(|V| + |V| \times |E_r| + |E_c|)$ where $E_r$ are transitive reduction arcs and $E_c$ are transitive closure arcs.

# Appendix C

# Testing the Validity of a Killing Function

In this chapter, we give an algorithm to test if a killing function $k$ of a DAG $G = (V, E, \delta, \delta_w, \delta_r)$is valid. We assume that the potential killing graph is constructed.

**Algorithm 11** boolean $valid\_killing\big(k, G = (V, E), PK(G)\big)$

$G_{\rightarrow k} = G$
**for all** value nodes $u \in G_{\rightarrow k}.V_R$ **do** {We add extended arcs within potential killing operations}
  **for all** nodes $v \in \Gamma^+_{PK(G)}.V_R$ **do**
    **if** $v \neq k(u)$ **then**
      $G_{\rightarrow k}.E_S = G_{\rightarrow k}.E_S \cup (u, v)$
    **end if**
  **end for**
**end for**
return Is_Acyclic($G_{\rightarrow k}$)

The complexity of algorithm 11 is the sum of:

- the construction of the extended graph $G_{\rightarrow k}$ in $O(|V_R| + E)$
- the test of the non existence of a circuit in $O(|V| + |G_{\rightarrow k}.E)| = O(|V| + |E| + |E_k|)$

So the whole complexity is dominated by $O(|V| + |E| + |E_k|)$

# Appendix D

# Building the bipartite decomposition for a DAG

We recall that a connected bipartite component is a triple $cb = (S_{cb}, T_{cb}, E_{cb})$ where:

- $S_{cb} \subseteq V_R$ is the parent values killed by the children in $T$;
- $T_{cb} \subseteq V$ are children that kill parents;
- $E_{cb} \subseteq E_{PK(G)}$ are the potential killing relation between $S_{cb}$ values and $T_{cb}$ nodes.

Algorithm 12 proceeds by selecting one value as an entry point for constructing a new bipartite component. Then, each child is added to the $T_{cb}$ set and each parent is inserted to the $S_{cb}$ set. This algorithm iterates until no new parent or child is found. The complexity of this algorithm is $O(|V| + |E_{PK(G)}|)$.

**Algorithm 12** Constructing the bipartite decomposition $\mathcal{B}(G)$

**Require:** $PK(G)$ of a DAG $G = (V, E, \delta, \delta_w, \delta_r)$
  $\mathcal{B}(G) = \phi$ {bipartite decomposition is initially empty}
  **for all** $u \in V_R$ **do** {initialization}
    visited[$u$]=false;
  **end for**
  **for all** $u \in V_R$ **do**
    **if** $\neg$ visited[$u$] **then** {we select one non visited value...}
      $cb = (\{u\}, \phi, \phi)$ {...to put it in $S_{cb}$}
      visited[$u$]=true;
      $T_{cb} = \Gamma^+_{PK(G)}(u)$
      $S = \phi$ {last $S_{cb}$}
      $T = \phi$ {last $T_{cb}$}
      **while** $(S \neq S_{cb}) \vee (T \neq T_{cb})$ **do** {grab all connected children with their parents}
        $S = S_{cb}$
        $T = T_{cb}$
        $S_{cb} = \cup_{t \in T_{cb}} \Gamma^-_{PK(G)}(t)$
        $T_{cb} = \cup_{s \in S_{cb}} \Gamma^+_{PK(G)}(s)$
      **end while**
      **for all** $s \in S_{cb}$ **do** {mark parent values as visited}
        visited[$s$]=true;
      **end for**
      $\mathcal{B}(G) = \mathcal{B}(G) \cup \{cb\}$
    **end if**
  **end for**

# Appendix E

# NP-Completness of Finding a SKS

The problem of finding a SKS can be formally generalize to a new set covering problem.

**Definition E.1 (set-weighted covering problem (SWC))** *:*
*Given a set $S$, and a family $\mathcal{E} = \{E \subseteq S\}$ of subsets of $S$ , and a function $\omega$ that assign to each edge $E \in \mathcal{E}$ a set $\omega(E)$ (called* **cost set***), could we find a covering $\mathcal{E}' \subseteq \mathcal{E}$ of $S$ such that*

1. *covering constraints*

$$\bigcup_{E \in \mathcal{E}'} = S$$

2. *minimizing the cost*

$$\min |\bigcup_{E \in \mathcal{E}'} \omega(E)|$$

Then, in the case of finding a SKS for a bipartite component $cb = (S, T, E_{cb})$, we have only to put

1. $\mathcal{E} = \left\{E_t \subseteq S \wedge t \in T / E_t = \{s \in S / (s,t) \in E_{cb}\}\right\}$
2. $\forall E_t \in \mathcal{E} : \omega(E_t) = \downarrow_{val} t$

**Theorem E.1** *SWC is NP-complete*

**Proof:**

We produce a sub-family of DAGs which can be reduced easily to the weighted covering problem. We choose a family of $\omega$ cost such that $\forall E, E' \in \mathcal{E} : \omega(E) \cap \omega(E') = \phi$. This means that the cost sets are completely disjoint. We can then write

$$|\bigcup_{E \in \mathcal{E}'} \omega(E)| = \sum_{E \in \mathcal{E}'} |\omega(E)|$$

This property permits us to reduce the SWC problem to the weighted covering problem easily. We first define a decision problem for this family of DAGs.

**Definition E.2 (SWC decision problem : dec(SWC))** *:*
*INSTANCE : a set $S$, and a family $\mathcal{E} = \{E \subseteq S\}$ of subsets of $S$, and a function $\omega$ that assign to each edge $E \in \mathcal{E}$ a set $\omega(E)$. Let be $j$ positive integer. The property of $\omega$ is*

$$\forall E, E' \in \mathcal{E} : \omega(E) \cap \omega(E') = \phi$$

*QUESTION : Does there exist a subset $\mathcal{E}' \subseteq \mathcal{E}$ such that :*

1. *the covering constraints*

$$\bigcup_{E \in \mathcal{E}'} = S$$

2. *the cost*

$$|\bigcup_{E \in \mathcal{E}'} \omega(E)| \leq j$$

We prove in the following that this decision problem is NP-complete. We reduce it to the weighted covering problem.

**Definition E.3 (Weighted Covering Problem (weicover))** *Given a set $X$, and a family $\mathcal{F} = \{F \subseteq X\}$ of subsets of $X$, and cost function $c$ that assign a positive integer cost to each subset $F \in \mathcal{F}$, find a covering $\mathcal{F}' \subseteq \mathcal{F}$ of $X$ such that:*

1. *the covering constraint*

$$\bigcup_{F \in \mathcal{F}'} F = X$$

2. *the minimization constraint*

$$\min \sum_{F \in \mathcal{F}'} c(F)$$

We define now a decision problem for the weicover problem.

**Definition E.4 (dec(weicover))** *:*
*INSTANCE : a set $X$, and a family $\mathcal{F} = \{F \subseteq X\}$ of subsets of $X$ , and cost function $c$ that assign a positive integer cost to each subset $F \in \mathcal{F}$. Let $j$ a positive integer.*
*QUESTION : Does there exist a covering $\mathcal{F}' \subseteq \mathcal{F}$ of $X$ such that*

1. *the covering constraint*

$$\bigcup_{F \in \mathcal{F}'} = X$$

2. *the minimum constraint*

$$\sum_{F \in \mathcal{F}'} c(F) \leq j$$

dec(weicover) is an NP-complete problem [Chv79]. It can be reduced easily to the minimum covering problem [CLR90] by putting $\forall F \in \mathcal{F} : c(F) = 1$. Now we prove that dec(SWC) is NP-complete.

**dec(SWC)** $\in NP$**:** Given an optimal covering $\mathcal{E}'$, we can construct the set $(\cup_{E\in\mathcal{E}'}\omega(E))$ in polynomial time. We have only to test if its cardinal is $\leq j$.

**dec(SWC) and dec(weicover) are equivalent** :

We associate to the instance of dec(SWC) an instance of dec(weicover) by putting

- $X = S$
- $\forall E \in \mathcal{E} : E = F$
- $\forall E \in \mathcal{E} : |\omega(E)| = c(F)$

Since the costs sets are disjoint, the cost of an optimal covering for dec(SWC) is then $\sum_{E\in\mathcal{E}'} |\omega(E)|$.

1. **dec(SWC)** $\Longrightarrow$ **dec(weicover)**

Suppose we have found an optimal cover $\mathcal{E}'$. . Then the subset $\mathcal{F}' = \{F/F = E \wedge E \in \mathcal{E}'\}$ is a solution for the weighted covering problem. This is because :

- Since $\mathcal{E}'$ satisfies the covering constraint for $S$, then $\mathcal{F}'$ is a cover for $X$.

$$\bigcup_{F\in\mathcal{F}'} = X$$

- since $\sum_{E\in\mathcal{E}'} |\omega(E)| \leq j$, then

$$\sum_{F\in\mathcal{F}'} c(F) \leq j$$

2. **dec(weicover)** $\Longrightarrow$ **dec(SKS)**

Now, suppose that we have a solution $\mathcal{F}'$ for the weighted covering problem. Then the subset $\mathcal{E}' = \{E/E = F \wedge F \in \mathcal{F}'\}$ is a solution for SKS. This is because :

1. Since $\mathcal{F}'$ satisfies the covering constraint for $X$, then $\mathcal{E}'$ satisfies the covering constraint for $S$

$$\bigcup_{E\in\mathcal{E}'} = S$$

2. since $\sum_{F\in\mathcal{F}'} |c(F)| \leq j$ then

$$\sum_{E\in\mathcal{E}'} |\omega(E)| \leq j$$

⌟

# Appendix F

# Studied Graphs

We give here the data dependence graphs treated in our experimentations, extracted from [Saw97]. Flow arcs are red arrows and serial arcs are black arrows. Values are represented with green circles, and other nodes with gray ones.

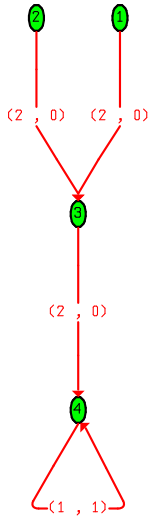



Figure F.1: lin-ddot

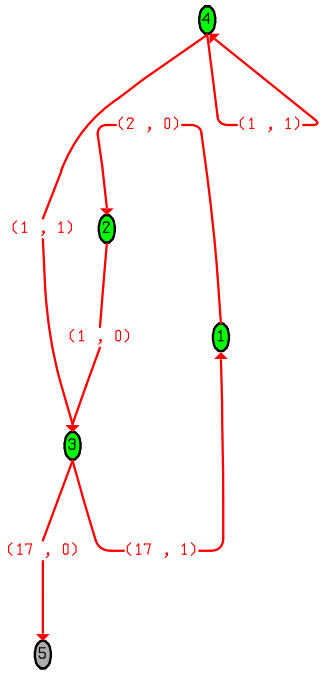



Figure F.2: spec-fppp

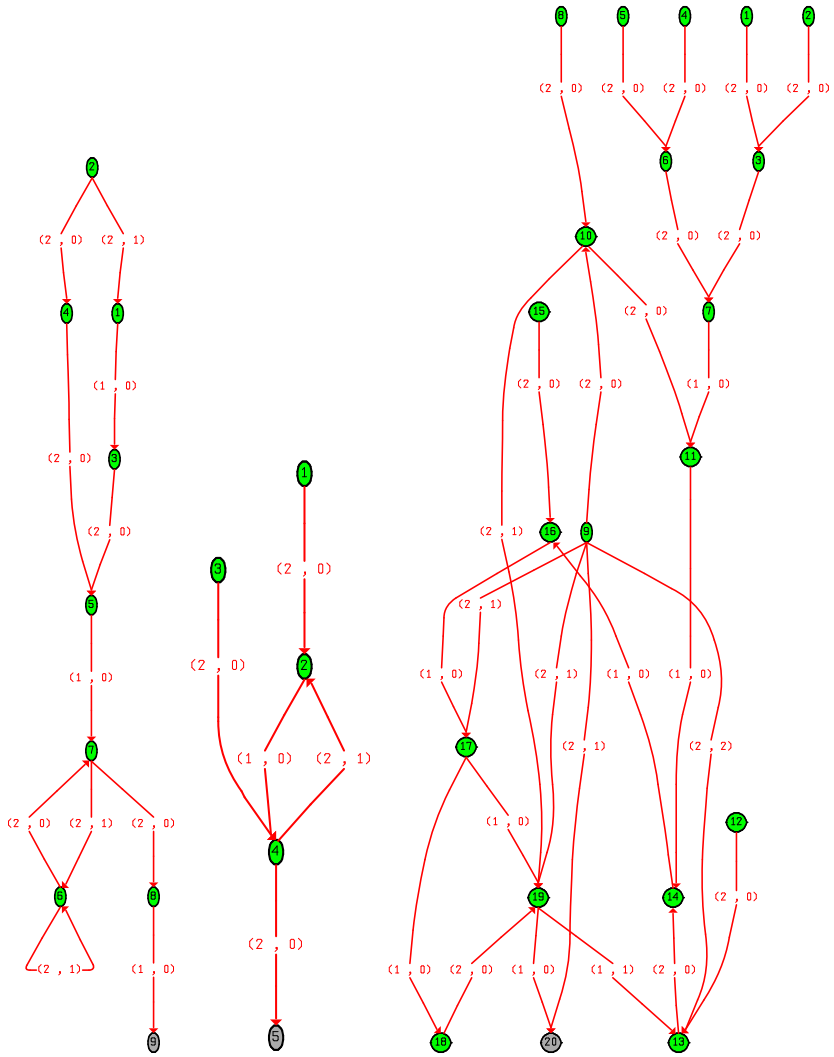


Figure F.3: Livermore: loop1, loop5, loop23 resp.

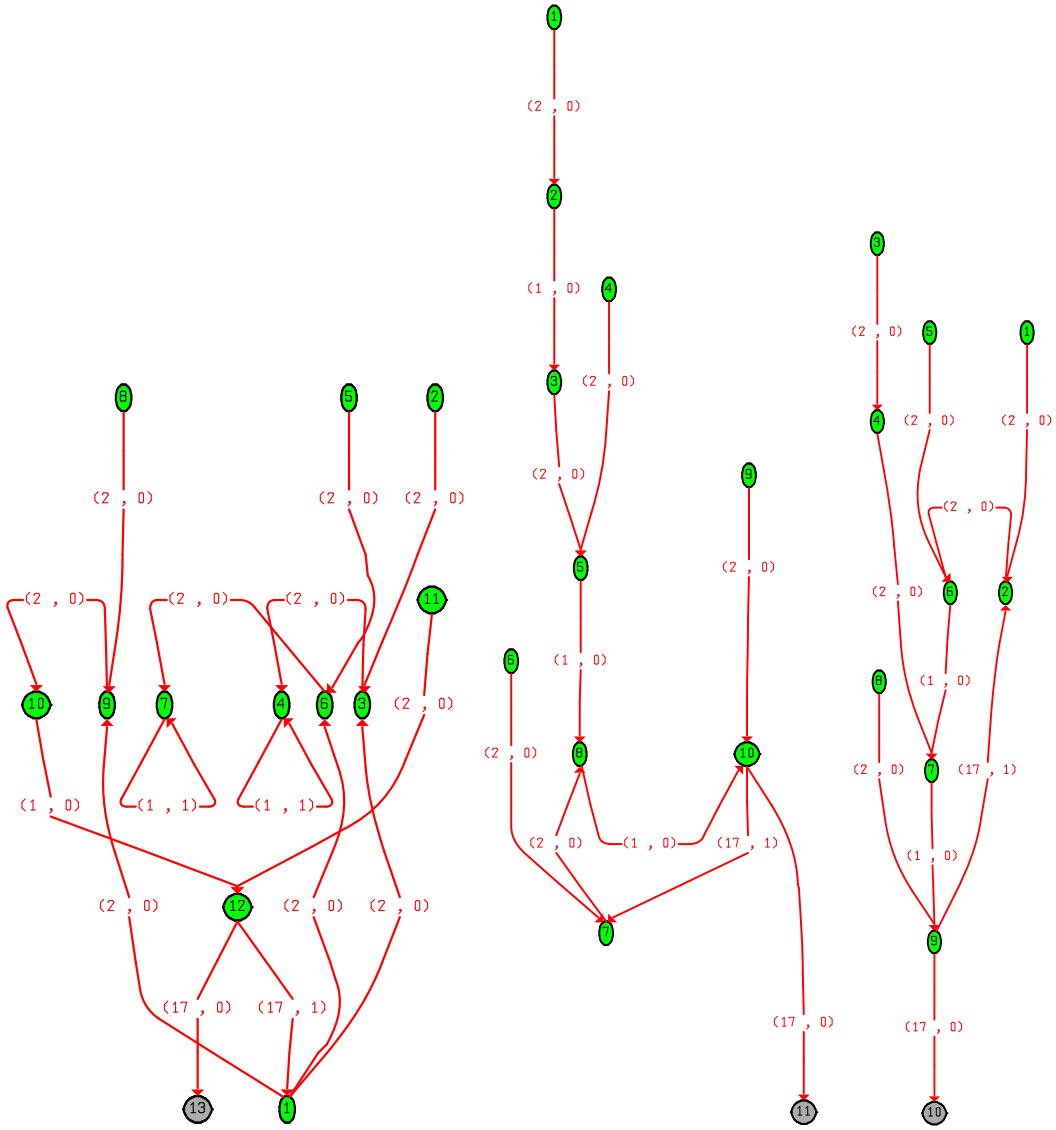



Figure F.4: spec-dod: loop1, loop3, loop2 resp.

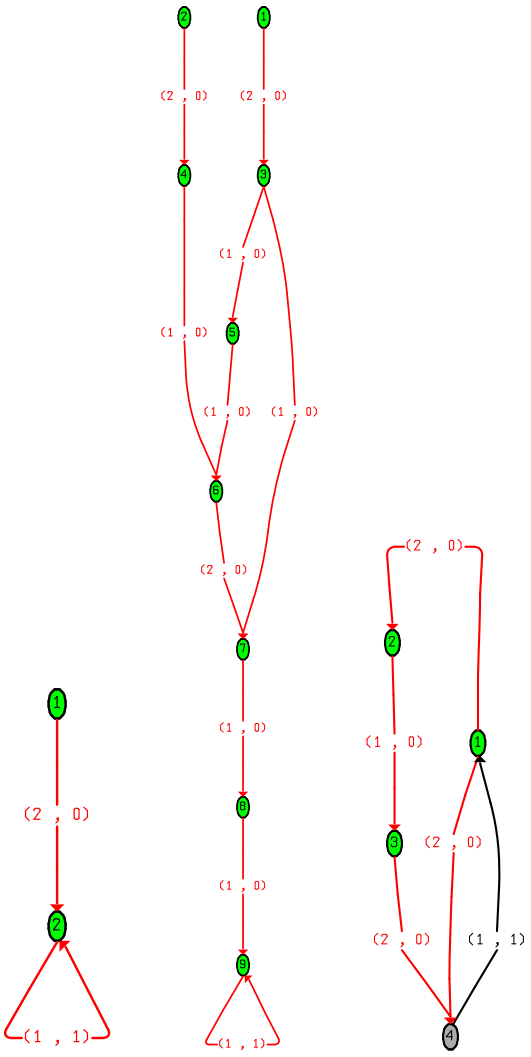



Figure F.5: spec-spice: loop1, loop2, loop3 resp.

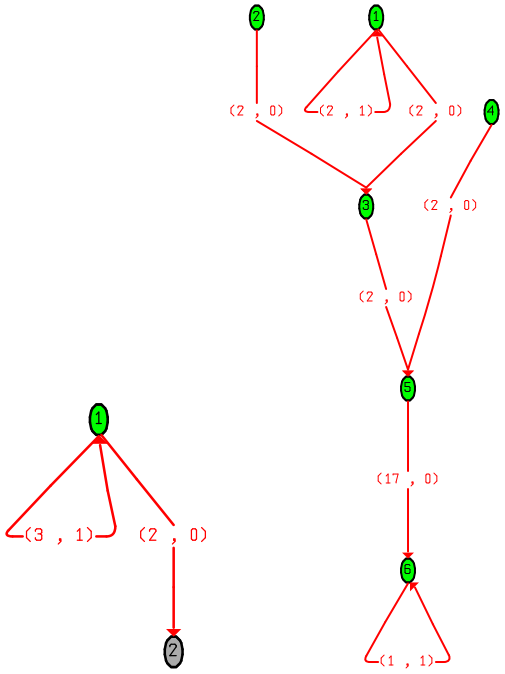



Figure F.6: spec-spice: loop5, loop6 resp.

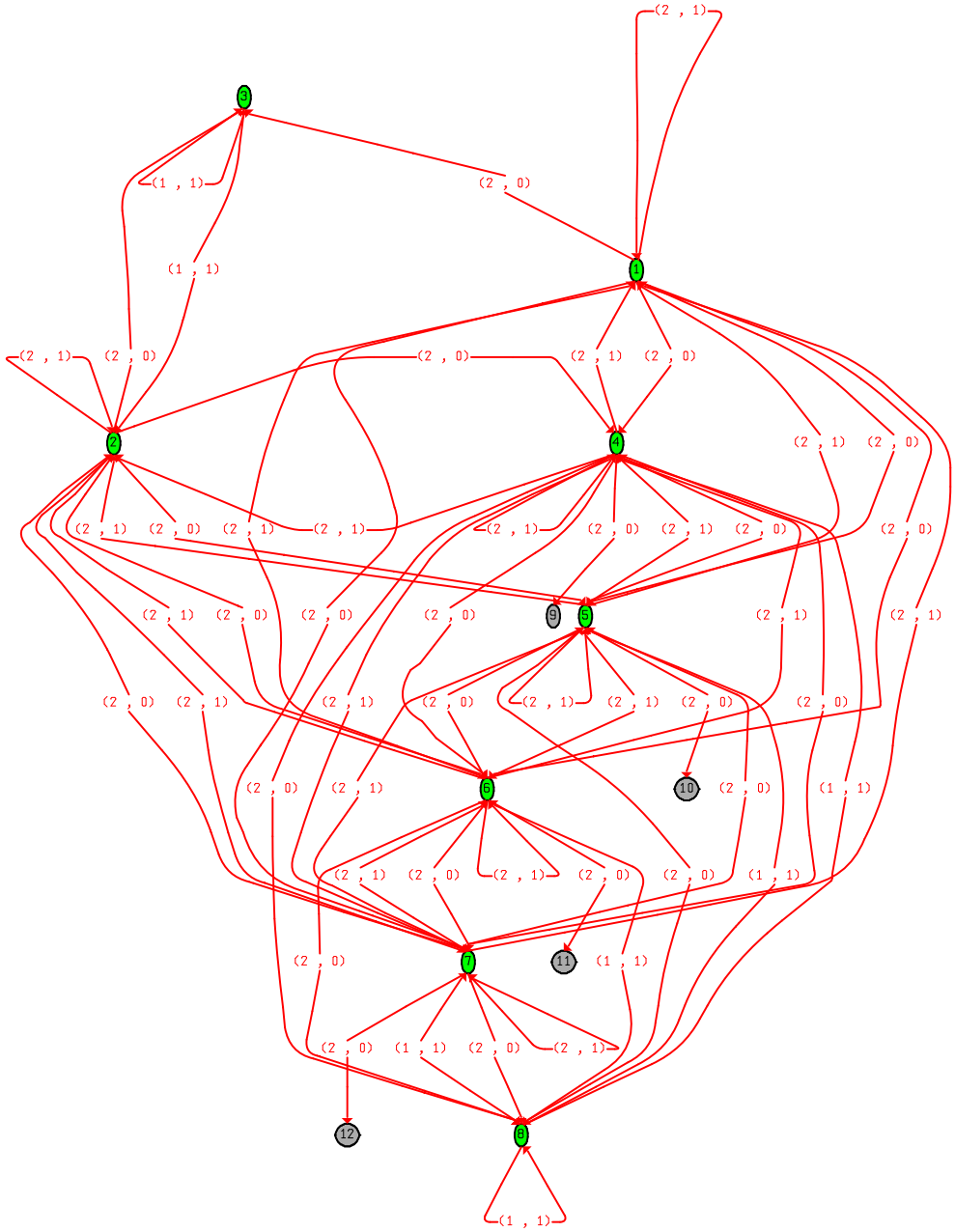



Figure F.7: spec-spice: loop4

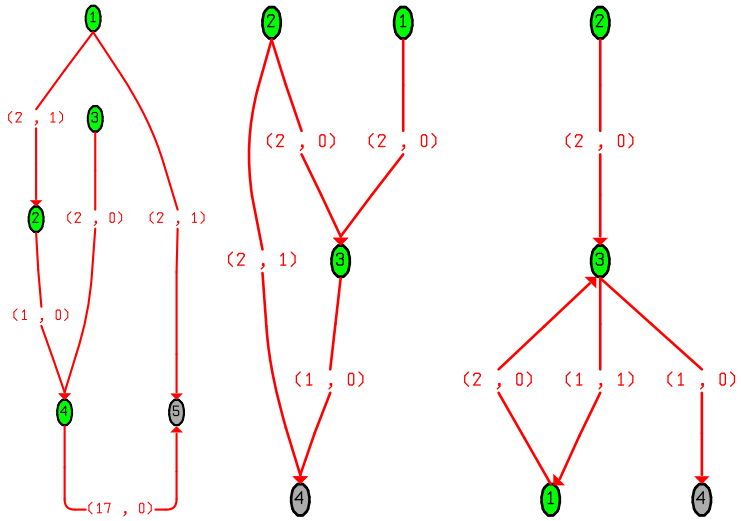



Figure F.8: spec-spice: loop7, loop8, loop10 resp.

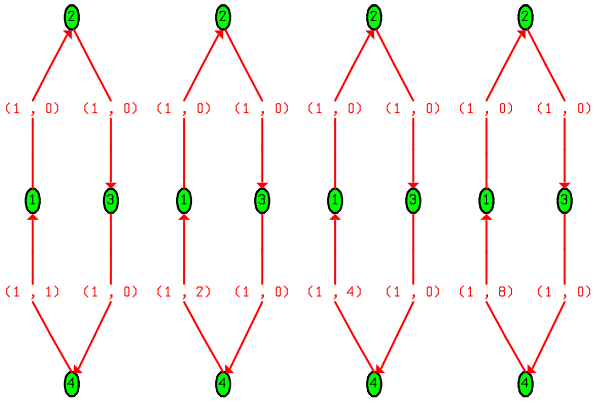



Figure F.9: cycles from whetstone: cycle1, cycle2, cycle4, cycle8 resp.

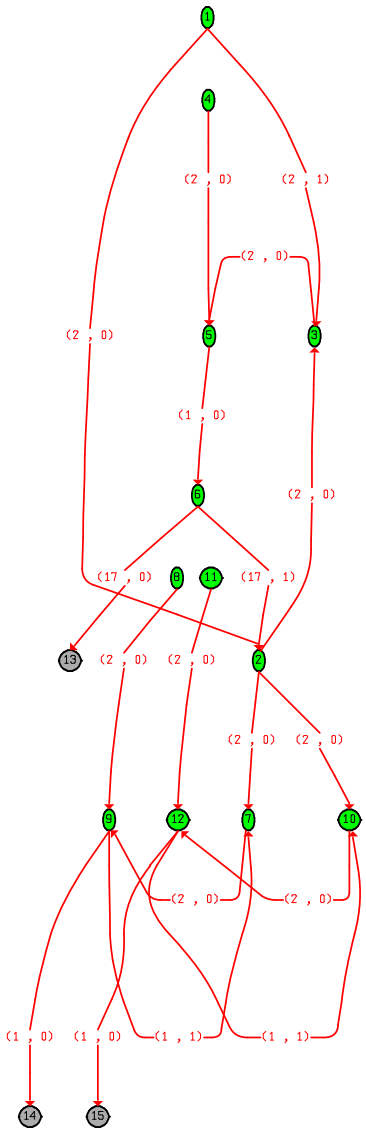



Figure F.10: spec-tomcatv: loop1

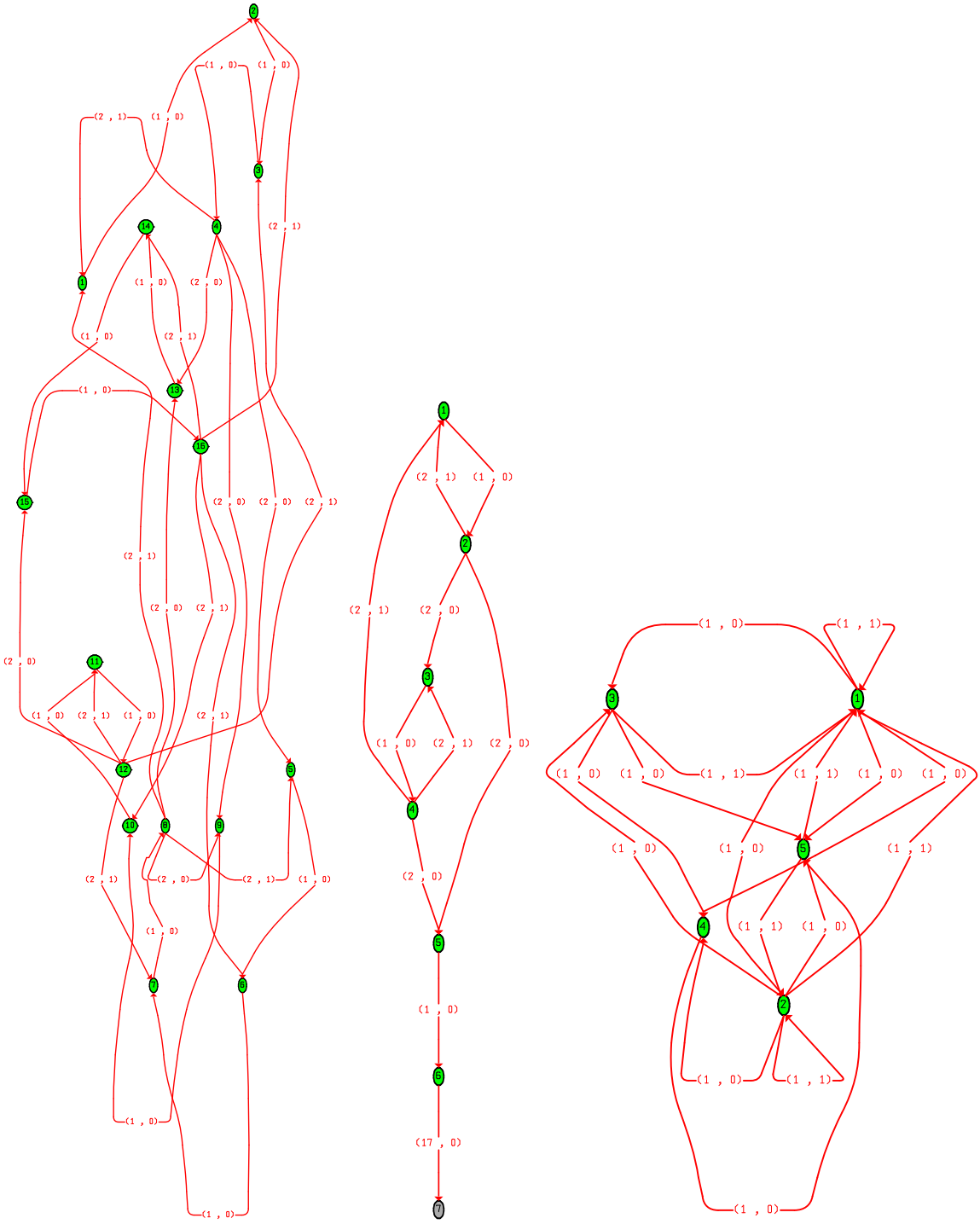


Figure F.11: whetstone: loop1, loop2, loop3 resp.

# Index

# Bibliography

[AKR91] Ajit Agrawal, Philip Klein, and R.Ravi. Ordering Problems Approximated : Register Sufficiency, Single Processor Scheduling and Interval Graph Completion. internal research report CS-91-18, Brown University, Providence, Rhode Island, March 1991.

[BEH91] David G. Bradlee, Susan J. Eggers, and Robert R. Henry. Integrating Register Allocation and Instruction Scheduling for RISCs. *ACM SIGPLAN Notices*, 26(4):122–131, April 1991.

[BGS92] D. Berson, R. Gupta, and M.L. Soffa. URSA: A Unified Resource Allocator for Registers and Functional Units in VLIW Architectures. Technical Report 92-21, University of Pittsburgh. Department of Computer Science, Pittsburgh, PA 15260, USA, December 1992.

[BGS93] D. Berson, R. Gupta, and M.L. Soffa. URSA: A unified ReSource allocator for registers and functional units in VLIW architectures. In *Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism*, pages 243–254, Orlando, Florida, January 1993.

[Bra94] Thomas S. Brasier. FRIGG: A New Approach to Combining Register Assignment and Instruction Scheduling. Master's thesis, Michigan Technological University, 1994.

[Chv79] V. Chvatal. A greedy heuristic for the set-covering problem. *Mathematics Of Operations Research*, 4(3):233–235, August 1979.

[CLR90] Thomas Cormen, Charles Eric Leiserson, and Ronald Rivest. *Introduction to Algorithms*. MIT Press/McGraw-Hill, Cambridge, Massachusetts, 1990.

[ES96] Christine Eisenbeis and Antoine Sawaya. Optimal Loop Parallelization under Register Constraints. Technical Report RR-2781, INRIA, January 1996. ftp://ftp.inria.fr/INRIA/publication/publi-ps-gz/RR/RR-2781.ps.gz.

[GAG94] R. Govindarajan, Erik R. Altman, and Guang R. Gao. Minimizing Register Requirements under Resource-Constrained Rate-Optimal Software Pipelining. In *MICRO27*, pages 85–94, December 1994.

[GH88] J. R. Goodman and W-C. Hsu. Code Scheduling and Register Allocation in Large Basic Blocks. In *Conference Proceedings 1988 International Conference on Supercomputing*, pages 442–452, St. Malo, France, July 1988.

[GS92] M. C. Golumbic and R. Shamir. Interval Graphs, Interval Orders and the Consistency of Temporal Events. In *Proceedings of Theory of Computing and Systems (ISTCS'92)*, volume 601 of *LNCS*, pages 32–42, Berlin, Germany, May 1992. Springer.

[Huf93] R. Huff. Lifetime-Sensitive Modulo Scheduling. In *PLDI 93*, pages 258–267, Albuquerque, New Mexico, June 1993.

[LGAV96] J. Llosa, A. Gonzalez, E. Ayguadé, and M. Valero. Swing Modulo Scheduling: A Lifetime-Sensitive Approach. In *PACT 96*, Boston, Massachusetts, October 20-23 1996.

[Llo96] Josep Llosa. *Reducing the Impact of Register Pressure on Software Pipelined Loops.* PhD thesis, Universitat Politecnica de Catalunya (Spain), 1996.

[LS93] I. Lemke and G. Sander. Visualization of Compiler Graphs. Technical Report Design report D 3.12.1-1, Universität des Saarlandes, 1993. ESPRIT Project 5399 Compare.

[LVA95] J. Llosa, M. Valero, and E. Ayguadé. Hypernode Reduction Modulo Scheduling. In *micro28*, pages 350–360, Boston, Massachusetts, November 1995.

[MN99] Kurt Mehlhorn and Stefan Näher. *LEDA: A Platform of Combinatorial and Geometric Computing.* Cambridge University Press, Cambridge, England, January 1999.

[Pin93] Schlomit S. Pinter. Register Allocation with Instruction Scheduling: A New Approach. *SIGPLAN Notices*, 28(6):248–257, June 1993.

[Saw97] Antoine Sawaya. *Pipeline Logiciel: Découplage et Contraintes de Registres.* PhD thesis, Université de Versailles Saint-Quentin-En-Yvelines, April 1997.

[SC96] Fermin Sanchez and Jordi Cortadella. RESIS: A New Methodology for Register Optimization in Software Pipelining. In *Proceedings of Second International Euro-Par Conference, Euro-Par'96*, Lyon, France, August 1996.

[SRM94] M. Schlansker, B. Rau, and S. Mahlke. Achieving High Levels of instruction-Level Parallelism with Reduced Hardware Complexity. Technical report, Hewlett Packard, 1994.

[WEJS94] Jian Wang, Christine Eisenbeis, Martin Jourdan, and Bogong Su. DEcomposed Software Pipelining: A new perspective and a new approach. *International Journal of Parallel Programming*, 22(3):351–373, June 1994.

[WKE95] Jian Wang, Andreas Krall, and M. Anton Ertl. Decomposed Software Pipelining with Reduced Register Requirement. In *Proceedings of the IFIP WG10.3 Working Conference on Parallel Architectures and Compilation Techniques, PACT95*, pages 277 – 280, Limassol, Cyprus, June 1995.

Unit´e de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unit´e de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unit´e de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unit´e de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unit´e de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399